

PYTHON BASICS



**A PRACTICAL INTRODUCTION
TO PYTHON 3**

FOURTH EDITION

BY THE REALPYTHON.COM TUTORIAL TEAM
DAVID AMOS, DAN BADER, JOANNA JABLONSKI, FLETCHER HEISLER

Python Basics: A Practical Introduction to Python 3

Real Python

Python Basics: A Practical Introduction to Python 3

Revised and Updated 4th Edition

David Amos, Dan Bader, Joanna Jablonski, Fletcher Heisler

Copyright © *Real Python* (realpython.com), 2012–2023

For online information and ordering of this and other books by *Real Python*, please visit realpython.com. For more information, please contact us at info@realpython.com.

ISBN: 9781775093329 (paperback)

ISBN: 9781775093336 (electronic)

Cover design by Aldren Santos

Additional editing and proofreading by Jacob Schmitt

“Python” and the Python logos are trademarks or registered trademarks of the Python Software Foundation, used by *Real Python* with permission from the Foundation.

Thank you for downloading this ebook. This ebook is licensed for your personal enjoyment only. This ebook may not be resold or given away to other people. If you would like to share this book with another person, please purchase an additional copy for each recipient. If you’re reading this book and did not purchase it, or if it was not purchased for your use only, then please return to realpython.com/pybasics-book and purchase your own copy. Thank you for respecting the hard work behind this book.

Updated 2023-12-12

What Pythonistas Say About *Python Basics: A Practical Introduction to Python 3*

“I love [the book]! The wording is casual, easy to understand, and makes the information flow well. I never feel lost in the material, and it’s not too dense so it’s easy for me to review older chapters over and over.

I’ve looked at over 10 different Python tutorials/books/online courses, and I’ve probably learned the most from Real Python!”

— **Thomas Wong**

“Three years later and I still return to my Real Python books when I need a quick refresher on usage of vital Python commands.”

— **Rob Fowler**

“I floundered for a long time trying to teach myself. I slogged through dozens of incomplete online tutorials. I snoozed through hours of boring screencasts. I gave up on countless crusty books from big-time publishers. And then I found Real Python.

The easy-to-follow, step-by-step instructions break the big concepts down into bite-sized chunks written in plain English. The authors never forget their audience and are consistently thorough and detailed in their explanations. I’m up and running now, but I constantly refer to the material for guidance.”

— **Jared Nielsen**

“I love the book because at the end of each particular lesson there are real world and interesting challenges. I just built a savings estimator that actually reflects my savings account – neat!”

— **Drew Prescott**

“As a practice of what you taught I started building simple scripts for people on my team to help them in their everyday duties. When my managers noticed that, I was offered a new position as a developer.

I know there is heaps of things to learn and there will be huge challenges, but I finally started doing what I really came to like.

Once again: MANY THANKS!”

— **Kamil**

“What I found great about the Real Python courses compared to others is how they explain things in the simplest way possible.

A lot of courses, in any discipline really, require the learning of a lot of jargon when in fact what is being taught could be taught quickly and succinctly without too much of it. The courses do a very good job of keeping the examples interesting.”

— **Stephen Grady**

“After reading the first Real Python course I wrote a script to automate a mundane task at work. What used to take me three to five hours now takes less than ten minutes!”

— **Brandon Youngdale**

“Honestly, throughout this whole process what I found was just me looking really hard for things that could maybe be added or improved, but this tutorial is amazing! You do a wonderful job of explaining and teaching Python in a way that people like me, a complete novice, could really grasp.

The flow of the lessons works perfectly throughout. The exercises truly helped along the way and you feel very accomplished when you finish up the book. I think you have a gift for making Python seem more attainable to people outside the programming world.

This is something I never thought I would be doing or learning and with a little push from you I am learning it and I can see that it will be nothing but beneficial to me in the future!”

— **Shea Klusewicz**

“The authors of the courses have NOT forgotten what it is like to be a beginner – something that many authors do – and assume nothing about their readers, which makes the courses fantastic reads. The courses are also accompanied by some great videos as well as plenty of references for extra learning, homework assignments and example code that you can experiment with and extend.

I really liked that there was always full code examples and each line of code had good comments so you can see what is doing what.

I now have a number of books on Python and the Real Python ones are the only ones I have actually finished cover to cover, and they are hands down the best on the market. If like me, you’re not a programmer (I work in online marketing) you’ll find these courses to be like a mentor due to the clear, fluff-free explanations! Highly recommended!”

— **Craig Addyman**

About the Authors

At [Real Python](#) you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The [realpython.com](#) website launched in 2012 and currently helps more than three million Python developers each month with free programming tutorials and in-depth learning resources.

Everyone who worked on this book is a *practitioner* with several years of professional experience in the software industry. Here are the members of the *Real Python* tutorial team who worked on *Python Basics*:

David Amos is the content technical lead for *Real Python*. After leaving academia in 2015, David worked in various technical positions as a programmer and data scientist. In 2019, David joined *Real Python* full time to pursue his passion for education. He lead the charge on rewriting and updating the *Python Basics* curriculum to Python 3.

Dan Bader is the owner and editor in chief of *Real Python* and the main developer of the [realpython.com](#) learning platform. Dan has been writing code for more than twenty years and holds a master's degree in computer science. He's the author of *Python Tricks*, a best-selling programming book for intermediate Python developers.

Joanna Jablonski is the executive editor of *Real Python*. She likes natural languages just as much as she likes programming languages. Her love for puzzles, patterns, and pesky little details led her to follow a career in translation. It was only a matter of time before she would fall in love with a new language: Python! She joined *Real Python* in 2018 and has been helping Pythonistas level up ever since.

Fletcher Heisler is the founder of Hunter2, where he teaches developers how to hack and secure modern web apps. As one of the founding members of *Real Python*, Fletcher wrote the first version of the Python curriculum this book is based on in 2012.

Contents

Contents	7
Foreword	12
1 Introduction	19
1.1 Why This Book?	20
1.2 About Real Python	22
1.3 How to Use This Book	23
1.4 Bonus Material and Learning Resources	24
2 Setting Up Python	28
2.1 A Note on Python Versions	29
2.2 Windows	30
2.3 macOS	33
2.4 Ubuntu Linux	36
3 Your First Python Program	41
3.1 Write a Python Program	42
3.2 Mess Things Up	46
3.3 Create a Variable	49
3.4 Inspect Values in the Interactive Window	54
3.5 Leave Yourself Helpful Notes	56
3.6 Summary and Additional Resources	59
4 Strings and String Methods	61
4.1 What Is a String?	62
4.2 Concatenation, Indexing, and Slicing	68

4.3	Manipulate Strings With Methods	78
4.4	Interact With User Input	84
4.5	Challenge: Pick Apart Your User’s Input	87
4.6	Working With Strings and Numbers	87
4.7	Streamline Your Prints	93
4.8	Find a String in a String	95
4.9	Challenge: Turn Your User Into a L33t H4xor	98
4.10	Summary and Additional Resources	99
5	Numbers and Math	101
5.1	Integers and Floating-Point Numbers	102
5.2	Arithmetic Operators and Expressions	106
5.3	Challenge: Perform Calculations on User Input	115
5.4	Make Python Lie to You	115
5.5	Math Functions and Number Methods	117
5.6	Print Numbers in Style	123
5.7	Complex Numbers	126
5.8	Summary and Additional Resources	129
6	Functions and Loops	131
6.1	What Is a Function, Really?	132
6.2	Write Your Own Functions	136
6.3	Challenge: Convert Temperatures	145
6.4	Run in Circles	146
6.5	Challenge: Track Your Investments	155
6.6	Understand Scope in Python	156
6.7	Summary and Additional Resources	161
7	Finding and Fixing Code Bugs	163
7.1	Use the Debug Control Window	164
7.2	Squash Some Bugs	170
7.3	Summary and Additional Resources	178
8	Conditional Logic and Control Flow	180
8.1	Compare Values	181
8.2	Add Some Logic	185
8.3	Control the Flow of Your Program	193

8.4	Challenge: Find the Factors of a Number	205
8.5	Break Out of the Pattern	206
8.6	Recover From Errors	210
8.7	Simulate Events and Calculate Probabilities	216
8.8	Challenge: Simulate a Coin Toss Experiment	222
8.9	Challenge: Simulate an Election	223
8.10	Summary and Additional Resources	223
9	Tuples, Lists, and Dictionaries	225
9.1	Tuples Are Immutable Sequences	226
9.2	Lists Are Mutable Sequences	236
9.3	Nesting, Copying, and Sorting Tuples and Lists	250
9.4	Challenge: List of lists	256
9.5	Challenge: Wax Poetic	257
9.6	Store Relationships in Dictionaries	259
9.7	Challenge: Capital City Loop	269
9.8	How to Pick a Data Structure	271
9.9	Challenge: Cats With Hats	272
9.10	Summary and Additional Resources	273
10	Object-Oriented Programming (OOP)	275
10.1	Define a Class	276
10.2	Instantiate an Object	280
10.3	Inherit From Other Classes	286
10.4	Challenge: Model a Farm	295
10.5	Summary and Additional Resources	296
11	Modules and Packages	297
11.1	Working With Modules	298
11.2	Working With Packages	309
11.3	Summary and Additional Resources	317
12	File Input and Output	319
12.1	Files and the File System	320
12.2	Working With File Paths in Python	323
12.3	Common File System Operations	332
12.4	Challenge: Move All Image Files to a New Directory .	349

12.5	Reading and Writing Files	350
12.6	Read and Write CSV Data	365
12.7	Challenge: Create a High Scores List	376
12.8	Summary and Additional Resources	377
13	Installing Packages With pip	378
13.1	Installing Third-Party Packages With pip	379
13.2	The Pitfalls of Third-Party Packages	389
13.3	Summary and Additional Resources	391
14	Creating and Modifying PDF Files	393
14.1	Extracting Text From a PDF	394
14.2	Extracting Pages From a PDF	401
14.3	Challenge: PdfFileSplitter Class	408
14.4	Concatenating and Merging PDFs	409
14.5	Rotating and Cropping PDF Pages	416
14.6	Encrypting and Decrypting PDFs	428
14.7	Challenge: Unscramble a PDF	432
14.8	Creating a PDF File From Scratch	432
14.9	Summary and Additional Resources	439
15	Working With Databases	441
15.1	An Introduction to SQLite	442
15.2	Libraries for Working With Other SQL Databases	454
15.3	Summary and Additional Resources	455
16	Interacting With the Web	457
16.1	Scrape and Parse Text From Websites	458
16.2	Use an HTML Parser to Scrape Websites	468
16.3	Interact With HTML Forms	474
16.4	Interact With Websites in Real Time	480
16.5	Summary and Additional Resources	484
17	Scientific Computing and Graphing	486
17.1	Use NumPy for Matrix Manipulation	487
17.2	Use Matplotlib for Plotting Graphs	498
17.3	Summary and Additional Resources	521

18 Graphical User Interfaces	522
18.1 Add GUI Elements With EasyGUI	523
18.2 Example App: PDF Page Rotator	536
18.3 Challenge: PDF Page Extraction Application	543
18.4 Introduction to Tkinter	544
18.5 Working With Widgets	548
18.6 Controlling Layout With Geometry Managers	573
18.7 Making Your Applications Interactive	592
18.8 Example App: Temperature Converter	602
18.9 Example App: Text Editor	607
18.10 Challenge: Return of the Poet	617
18.11 Summary and Additional Resources	618
19 Final Thoughts and Next Steps	620
19.1 Free Weekly Tips for Python Developers	621
19.2 Python Tricks: The Book	621
19.3 Real Python Video Course Library	622
19.4 Acknowledgements	623

Foreword

Hello, and welcome to *Python Basics: A Practical Introduction to Python 3*. I hope you're ready to learn why so many professional and hobbyist developers are drawn to Python and how you can begin using it on your own projects, small and large, right away.

This book is targeted at beginners who either know a little programming but not the Python language and ecosystem or are starting fresh with no programming experience whatsoever.

If you don't have a computer science degree, don't worry. David, Dan, Joanna, and Fletcher will guide you through the important computing concepts while teaching you the Python basics and, just as importantly, skipping the unnecessary details at first.

Python Is a Full-Spectrum Language

When learning a new programming language, you don't yet have the experience to judge how well it will serve you in the long run. If you're considering learning Python, let me assure you that this is a good choice. One key reason is that Python is a **full-spectrum** language.

What do I mean by this? Some languages are very good for beginners. They hold your hand and make programming super easy. We can go to the extreme and look at visual languages such as Scratch.

In Scratch, you get blocks that represent programming concepts like variables, loops, method calls, and so on, and you drag and drop them on a visual surface. Scratch may be easy to get started with for sim-

ple programs, but you cannot build professional applications with it. Name one Fortune 500 company that powers its core business logic with Scratch.

Come up empty? Me too, because that would be insanity.

Other languages are incredibly powerful for expert developers. The most popular one in this category is likely C++ and its close relative, C. Whichever web browser you used today was likely written in C or C++. Your operating system running that browser was very likely also built with C/C++. Your favorite first-person shooter or strategy video game? You nailed it: C/C++.

You can do amazing things with these languages, but they are wholly unwelcoming to newcomers looking for a gentle introduction.

You might not have read a lot of C++ code. It can almost make your eyes burn. Here's an example, a real albeit complex one:

```
template <typename T>
_Defer<void(*(PID<T>, void (T::*)(void)))>
    (const PID<T>&, void (T::*)(void))>
defer(const PID<T>& pid, void (T::*method)(void))
{
    void (*dispatch)(const PID<T>&, void (T::*)(void)) =
        &process::template dispatch<T>;
    return std::bind(dispatch, pid, method);
}
```

Please, just no.

Both Scratch and C++ are decidedly *not* what I would call full-spectrum languages. With Scratch, it's easy to start, but you have to switch to a "real" language to build real applications. Conversely, you can build real apps with C++, but there's no gentle on-ramp. You dive headfirst into all the complexity of the language, which exists to support these rich applications.

Python, on the other hand, is special. It is a full-spectrum language. We often judge the simplicity of a language based on the Hello, World test. That is, what syntax and actions are necessary to get the language to output Hello, World to the user? In Python, it couldn't be simpler:

```
print("Hello, World")
```

That's it! However, I find this an unsatisfying test.

The Hello, World test is useful but really not enough to show the power or complexity of a language. Let's try another example. Not everything here needs to make total sense—just follow along to get the Zen of it. The book covers these concepts and more as you go through. The next example is certainly something you could write as you get near the end of the book.

Here's the new test: What would it take to write a program that accesses an external website, downloads the content to your app in memory, then displays a subsection of that content to the user? Let's try that experiment using Python 3 with the help of the requests package (which needs to be installed—more on that in chapter 13):

```
import requests
resp = requests.get("http://olympus.realpython.org")
html = resp.text
print(html[86:132])
```

Incredibly, that's it! When run, the program outputs something like this:

```
<h2>Please log in to access Mount Olympus:</h2>
```

This is the easy, getting-started side of the Python spectrum. A few trivial lines can unleash incredible power. Because Python has access to so many powerful but well-packaged libraries, such as requests, it's often described as *having batteries included*.

So there you have a simple yet powerful starter example. On the real-world side of things, many incredible applications have been written in Python as well.

YouTube, the world's most popular video streaming site, is written in Python and processes more than a million requests per second. Instagram is another example of a Python application. Closer to home, we even have realpython.com and my sites, such as talkpython.fm.

This full-spectrum aspect of Python means that you can start with the basics and adopt more advanced features as your application demands grow.

Python Is Popular

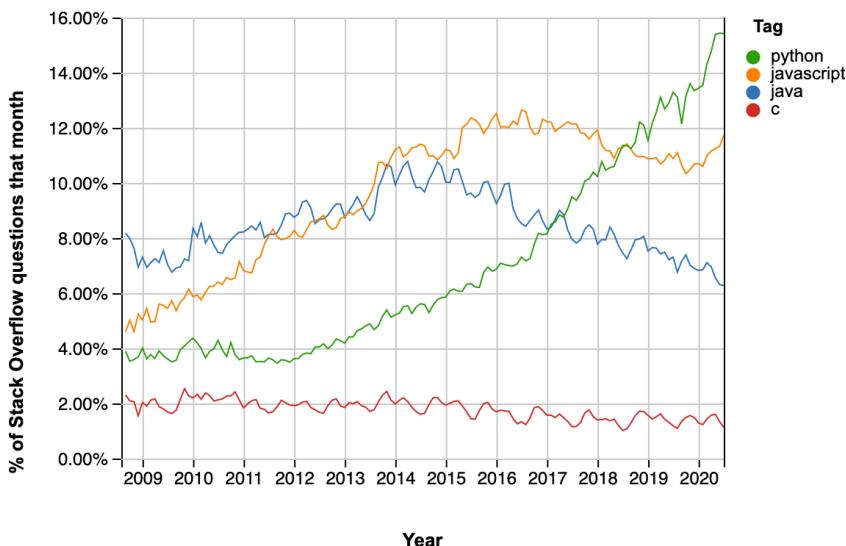
You might have heard that Python is popular. It may seem that it doesn't really matter how popular a language is so long as you can build the app you want to build with it.

But, for better or worse, the popularity of a programming language is a strong indicator of the quality of libraries you'll have available as well the number of job openings you'll find. In short, you should tend to gravitate toward more popular technologies as there will be more choices and integrations available.

So, is Python actually that popular? Yes it is. You'll find a lot of hype and hyperbole, but there are plenty of stats backing this claim. Let's look at some analytics presented by [stack overflow.com](http://stackoverflow.com), a popular question-and-answer site for programmers.

Stack Overflow runs a site called Stack Overflow Trends where you can look at the trends for various technologies by tag. When you compare

Python to the other likely candidates you could pick to learn programming, you'll see one is unlike the others:



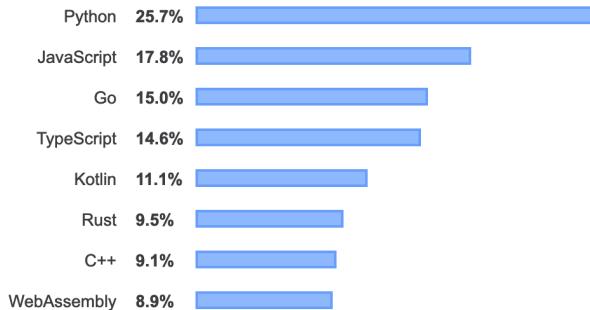
You can explore this chart and create similar charts to this one over at insights.stackoverflow.com/trends.

Notice the incredible growth of Python compared to the flat or even downward trend of the other usual candidates! If you're betting your future on the success of a given technology, which one would you choose from this list?

That's just one chart—what does it really tell us? Well, let's look at another. Stack Overflow does a yearly survey of developers. It's comprehensive and very well done. You can find the full 2020 results at insights.stackoverflow.com/survey/2020.

From that writeup, I'd like to call your attention to a section titled “[Most Loved, Dreaded, and Wanted Languages](#).” In the “Most Wanted” section, you'll find data on the share of “developers who are not developing with the language or technology but have expressed interest in developing with it.”

Again, in the graph below, you'll see that Python is topping the charts and is well above even second place:



If you agree with me that the relative popularity of a programming language matters, then Python is clearly a good choice.

We Don't Need You to Be a Computer Scientist

One other point that I want to emphasize as you start your Python learning journey is that we don't need you to be a computer scientist. If that's your goal, then great. Learning Python is a powerful step in that direction. But the invitation to learn programming is often framed as "We have all these developer jobs going unfilled! We need software developers!"

That may or may not be true. But, more importantly, programming (even a little programming) can be a personal superpower for you.

To illustrate this idea, suppose you are a biologist. Should you drop out of biology and get a job as a front-end web developer? Probably not. But skills such as the one I opened this foreword with, using requests to get data from the Web, can be incredibly powerful for you as a biologist.

Rather than manually exporting and scraping data from the Web or from spreadsheets, you can use Python to scrape thousands of data sources or spreadsheets in the time it takes you to do just one man-

ually. Python skills can take your *biology power* and amplify it well beyond your colleagues' to make it your *superpower*.

Dan and Real Python

Finally, let me leave you with a comment on your authors. Dan Bader and the other *Real Python* authors work day in and day out to bring clear and powerful explanations of Python concepts to all of us via realpython.com.

They have a unique view into the Python ecosystem and are keyed into what beginners need to know.

I'm confident leaving you in their hands on this Python journey. Go forth and learn this amazing language using this great book. Most importantly, remember to have fun!

— **Michael Kennedy**, Founder of Talk Python ([@mkennedy](https://twitter.com/mkennedy))

Chapter 1

Introduction

Welcome to *Real Python's Python Basics* book, fully updated for Python 3.9! In this book, you'll learn real-world Python programming techniques, illustrated with useful and interesting examples.

Whether you're a new programmer or a professional software developer looking to dive into a new language, this book will teach you all the practical Python that you need to get started on projects of your own.

No matter what your ultimate goals may be, if you work with a computer at all, then you'll soon be finding endless ways to improve your life by automating tasks and solving problems through Python programs that you create.

But what's so great about Python as a programming language? For one, Python is open source freeware, meaning you can download it for free and use it for any purpose, commercial or not.

Python also has an amazing community that has built a number of useful tools that you can use in your own programs. Need to work with PDF documents? There's a comprehensive tool for that. Want to collect data from web pages? No need to start from scratch!

Python was built to be easier to use than other programming languages. It's usually much easier to read Python code and much faster to write code in Python than in other languages.

For instance, here's some basic code written in C, another commonly used programming language:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, World\n");
}
```

All the program does is show the text `Hello, World` on the screen. That was a lot of work to output one phrase! Here's the same program written in Python:

```
print("Hello, World")
```

That's pretty simple, right? The Python code is faster to write and easier to read. We find that it looks friendlier and more approachable, too!

At the same time, Python has all the functionality of other languages and more. You might be surprised by how many professional products are built on Python code: Instagram, YouTube, Reddit, Spotify, to name just a few.

Python is not only a friendly and fun language to learn, but it also powers the technology behind multiple world-class companies and offers fantastic career opportunities for any programmer who masters it.

1.1 Why This Book?

Let's face it: there's an overwhelming amount of information about Python on the Internet. But many beginners studying on their own have trouble figuring out *what* to learn and *in what order* to learn it.

You may be asking yourself, What should I learn about Python in the beginning to get a strong foundation? If so, then this book is for you, no matter if you're a complete beginner or if you've already dabbled in Python or other languages.

Python Basics is written in plain English and breaks down the core concepts that you really need to know into bite-sized chunks. This means you'll learn enough to be dangerous with Python, fast.

Instead of just going through a boring list of language features, you'll see exactly how the different building blocks fit together and what's involved in building real applications and scripts with Python.

Step by step, you'll master fundamental Python concepts that will help you get started on your journey toward learning Python.

Many programming books try to cover every last possible variation of every command, which makes it easy for readers to get lost in the details. This approach is great if you're looking for a reference manual, but it's a horrible way to learn a programming language. Not only do you spend most of your time cramming things into your head that you'll never use, but you also don't have any fun!

This book is built on the 80/20 principle, which suggests that you can learn most of what you need to know by focusing on a few crucial concepts. We'll cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to everyday problems.

This way, we guarantee that you will:

- Learn useful programming techniques quickly
- Spend less time struggling with unimportant complications
- Find more practical uses for Python in your own life
- Have more fun in the process

Once you've mastered the material in this book, you will have gained a strong enough foundation that venturing out on your own into more advanced territory will be a breeze.

What you'll learn here is based on the first part of the original *Real Python Course* initially released in 2012. Over the years, this Python curriculum has been battle-tested by thousands of Pythonistas, data scientists, and developers working for companies big and small, including Amazon, Red Hat, and Microsoft.

For *Python Basics*, we've thoroughly expanded, refined, and updated the material so you can build your Python skills quickly and efficiently.

1.2 About Real Python

At *Real Python*, you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than three million Python developers each month with books, programming tutorials, and other in-depth learning resources.

Everyone who worked on this book is a *Python practitioner* recruited from the Real Python team with several years of professional experience in the software industry.

Here's where you can find *Real Python* on the Web:

- realpython.com
- [@realpython on Twitter](https://twitter.com/realpython)
- [The *Real Python Newsletter*](https://realpython.com/newsletter)
- [The *Real Python Podcast*](https://realpython.com/podcast)

1.3 How to Use This Book

The first half of this book is a quick but thorough overview of all the Python fundamentals. You don't need any prior experience with programming to get started. The second half is focused on finding practical solutions to interesting, real-world coding problems.

If you're a beginner, then we recommend that you go through the first half of this book from beginning to end. The second half covers topics that don't overlap as much, so you can jump around more easily, but the chapters do increase in difficulty as you go along.

If you're a more experienced programmer, then you may find yourself heading toward the second part of the book right away. But don't neglect getting a strong foundation in the basics first, and be sure to fill in any knowledge gaps along the way.

Most sections within a chapter are followed by **review exercises** to help you make sure that you've mastered all the topics covered. There are also a number of **code challenges**, which are more involved and usually require you to tie together several different concepts from previous chapters.

The practice files that accompany this book also include full solutions to the challenges as well as some of the trickier exercises. But to get the most out of the material, you should try your best to solve the challenge problems on your own before looking at the example solutions.

If you're completely new to programming, then you may want to supplement the first few chapters with additional practice. We recommend working through the entry-level tutorials available for free at realpython.com to make sure you're on solid footing.

If you have any questions or feedback about the book, you're always welcome to [contact us](#) directly.

Learning by Doing

This book is all about learning by doing, so be sure to *actually type in* the code snippets you encounter in the book. For best results, we recommend that you avoid copying and pasting the code examples.

You'll learn the concepts better and pick up the syntax faster if you type out each line of code yourself. Plus, if you screw up—which is totally normal and happens to all developers on a daily basis—the simple act of correcting typos will help you learn how to debug your code.

Try to complete the review exercises and code challenges on your own before getting help from outside resources. With enough practice, you'll master this material—and have fun along the way!

How Long Will It Take to Finish This Book?

If you're already familiar with a programming language, then you could finish this book in as little as thirty-five to forty hours. If you're new to programming, then you may need to spend up to one hundred hours or more.

Take your time and don't feel like you have to rush. Programming is a super-rewarding but complex skill to learn. Good luck on your Python journey. We're rooting for you!

1.4 Bonus Material and Learning Resources

This book comes with a number of free bonus resources and downloads that you can access online at the link below. We're also maintaining an errata list with corrections there:

realpython.com/python-basics/resources

Interactive Quizzes

Most chapters in this book come with a free online quiz to check your learning progress. You can access the quizzes using the links provided at the end of the chapter. The quizzes are hosted on the *Real Python* website and can be viewed on your phone or computer.

Each quiz takes you through a series of questions related to a particular chapter in the book. Some of them are multiple choice, some will ask you to type in an answer, and some will require you to write actual Python code. As you make your way through each quiz, it will keep score of which questions you answered correctly.

At the end of the quiz, you'll receive a grade based on your result. If you don't score 100 percent on your first try, don't fret! These quizzes are meant to challenge you. It's expected that you'll go through them several times, improving your score with each run.

Exercises Code Repository

This book has an accompanying [code repository on the Web](#) containing example source code as well as the answers to exercises and code challenges. The repository is broken up by chapter, so you can check your code against the solutions provided by us after you finish each chapter. Here's the link:

realpython.com/python-basics/exercises

Note

The code found in this book has been tested with Python 3.9 on Windows, macOS, and Linux.

Example Code License

The example Python scripts associated with this book are licensed under a [Creative Commons Public Domain \(CCo\) License](#). This means that you're welcome to use any portion of the code for any purpose in your own programs.

Formatting Conventions

Code blocks will be used to present example code:

```
# This is Python code:
print("Hello, World")
```

Terminal commands follow the Unix format:

```
$ # This is a terminal command:
$ python hello-world.py
```

(The dollar signs are not part of the command.)

Monospace text will be used to denote a filename: `hello-world.py`.

Bold text will be used to denote a new or important term.

Keyboard shortcuts will be formatted as follows: `Ctrl + S`

Menu shortcuts will be formatted as follows: `File > New File`

Notes and important information will be highlighted as follows:

Note

This is a note filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Feedback and Errata

We welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Did you find an error in the text or code? Did we leave out a topic that you'd love to know more about?

We're always looking to improve our teaching materials. Whatever the reason, please send in your feedback at the link below:

realpython.com/python-basics/feedback

Chapter 2

Setting Up Python

This book is about programming computers with Python. You could read this book from cover to cover without ever touching a keyboard, but you'd miss out on the fun part—coding!

To get the most out of this book, you need a computer with Python installed on it and a way to create, edit, and save Python code files.

In this chapter, you'll learn how to:

- Install the latest version of Python 3 on your computer
- Open **IDLE**, Python's built-in **Integrated Development and Learning Environment**

Let's get started!

2.1 A Note on Python Versions

Many operating systems, including macOS and Linux, come with Python preinstalled. The version of Python that comes with your operating system is called the **system Python**.

The system Python is used by your operating system and is usually out of date. It's essential that you have the most recent version of Python so that you can successfully follow along with the examples in this book.

Important

Do not attempt to uninstall the system Python!

You can have multiple versions of Python installed on your computer. In this chapter, you'll install the latest version of Python 3 alongside any system Python that may already exist on your machine.

Note

Even if you already have Python 3.9 installed, it's still a good idea to skim this chapter to double-check that your environment is set up for following along with this book.

This chapter is split into three sections: Windows, macOS, and Ubuntu Linux. Find the section for your operating system and follow the steps to get set up, then skip ahead to the next chapter.

If you have a different operating system, then check out *Real Python's* “[Python 3 Installation & Setup Guide](#)” to see if your OS is covered. Readers on tablets and mobile devices can refer to the “[Online Python Interpreters](#)” section for some browser-based options.

2.2 Windows

Follow these steps to install Python 3 and open IDLE on Windows.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

Windows doesn't typically come with a system Python. Fortunately, installation involves little more than downloading and running the Python installer from the [Python.org website](https://www.python.org).

Step 1: Download the Python 3 Installer

Open a web browser and navigate to the following URL:

<https://www.python.org/downloads/windows/>

Click *Latest Python 3 Release - Python 3.x.x* located beneath the “Python Releases for Windows” heading near the top of the page. As of this writing, the latest version was Python 3.9.

Then scroll to the bottom and click *Windows x86-64 executable installer* to start the download.

Note

If your system has a 32-bit processor, then you should choose the 32-bit installer. If you aren't sure if your computer is 32-bit or 64-bit, stick with the 64-bit installer mentioned above.

Step 2: Run the Installer

Open your Downloads folder in Windows Explorer and double-click the file to run the installer. A dialog that looks like the following one will appear:



It's okay if the Python version you see is greater than 3.9.1 as long as the version is not less than 3.

Important

Make sure you select the box that says *Add Python 3.x to PATH*. If you install Python without selecting this box, then you can run the installer again and select it.

Click **Install Now** to install Python 3. Wait for the installation to finish, then continue to open IDLE.

Open IDLE

You can open IDLE in two steps:

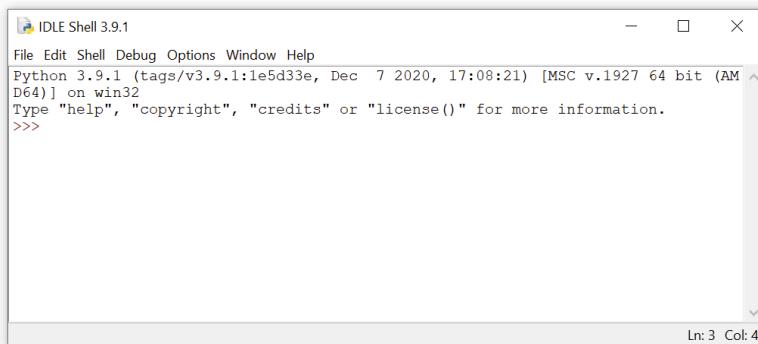
1. Click the Start menu and locate the Python 3.9 folder.
2. Open the folder and select *IDLE (Python 3.9)*.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

The `>>>` symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

2.3 macOS

Follow these steps to install Python 3 and open IDLE on macOS.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

To install the latest version of Python 3 on macOS, download and run the official installer from the [Python.org](https://www.python.org) website.

Step 1: Download the Python 3 Installer

Open a web browser and navigate to the following URL:

<https://www.python.org/downloads/mac-osx/>

Click *Latest Python 3 Release - Python 3.x.x* located beneath the “Python Releases for Mac OS X” heading near the top of the page. As of this writing, the latest version was Python 3.9.

Then scroll to the bottom of the page and click *macOS 64-bit installer* to start the download.

Step 2: Run the Installer

Open Finder and double-click the downloaded file to run the installer. A dialog box that looks like the following will appear:



Press **Continue** a few times until you are asked to agree to the software license agreement. Then click **Agree**.

You'll be shown a window that tells you where Python will be installed and how much space it will take. You most likely don't want to change the default location, so go ahead and click **Install** to start the installation.

When the installer is finished copying files, click **Close** to close the installer window.

Open IDLE

You can open IDLE in three steps:

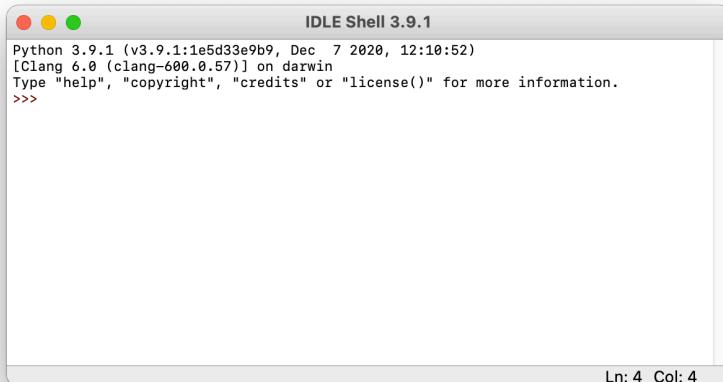
1. Open Finder and click *Applications*.
2. Double-click the Python 3.9 folder.
3. Double-click the IDLE icon.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

The `>>>` symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

2.4 Ubuntu Linux

Follow these steps to install Python 3 and open IDLE on Ubuntu Linux.

Important

The code in this book is tested only against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running some of the code examples.

Install Python

There's a good chance that your Ubuntu distribution already has Python installed, but it probably won't be the latest version, and it may be Python 2 instead of Python 3.

To find out what version(s) you have, open a terminal window and try the following commands:

```
$ python --version  
$ python3 --version
```

One or more of these commands should respond with a version, as below:

```
$ python3 --version  
Python 3.9.1
```

Your version number may vary. If the version shown is Python 2.x or a version of Python 3 that is less than 3.9, then you want to install the latest version. How you install Python on Ubuntu depends on which version of Ubuntu you're running. You can determine your local Ubuntu version by running the following command:

```
$ lsb_release -a  
No LSB modules are available.  
Distributor ID: Ubuntu  
Description:    Ubuntu 18.04.1 LTS  
Release:        18.04  
Codename:       bionic
```

Look at the version number next to `Release` in the console output, and follow the corresponding instructions below.

Ubuntu 18.04 or Greater

Ubuntu version 18.04 does not come with Python 3.9 by default, but it is in the Universe repository. You can install it with the following commands in the Terminal application:

```
$ sudo apt-get update  
$ sudo apt-get install python3.9 idle-python3.9 python3-pip
```

Note that because the Universe repository is usually behind the Python release schedule, you may not get the latest version of Python 3.9. However, any version of Python 3.9 will work for this book.

Ubuntu 17 and Lower

For Ubuntu versions 17 and lower, Python 3.9 is not in the Universe repository. You need to get it from a Personal Package Archive (PPA). To install Python from the [deadsnakes PPA](#), run the following commands in the Terminal application:

```
$ sudo add-apt-repository ppa:deadsnakes/ppa  
$ sudo apt-get update  
$ sudo apt-get install python3.9 idle-python3.9 python3-pip
```

You can check that the correct version of Python was installed by running `python3 --version`. If you see a version number less than 3.9, then you may need to type `python3.9 --version`. Now you can open IDLE and get ready to write your first Python program.

Open IDLE

You can open IDLE from the command line by typing the following:

```
$ idle-python3.9
```

On some Linux installations, you can open IDLE with the following shortened command:

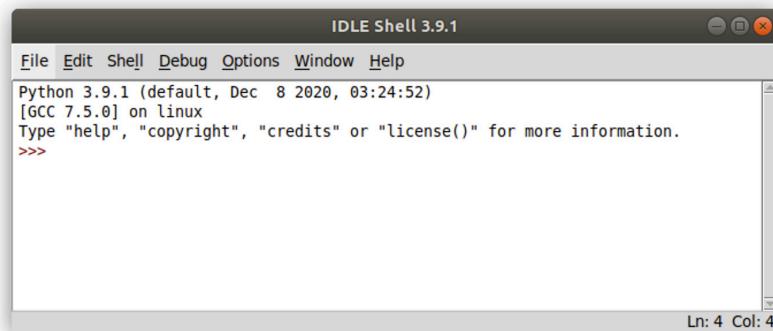
```
$ idle3
```

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in Python code and execute it immediately. It's a great way to get started with Python!

Note

While you're free to use a code editor other than IDLE if you prefer, note that some chapters, especially chapter 7, "Finding and Fixing Code Bugs," do contain material specific to IDLE.

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.9, then you may need to revisit the installation instructions in the previous section.

Important

If you open IDLE with the `idle3` command and see a version less than 3.9 displayed in the Python shell window, then you'll need to open IDLE with the `idle-python3.9` command.

The `>>>` symbol that you see in the IDLE window is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-setup

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to chapter 3.

Chapter 3

Your First Python Program

Now that you have the latest version of Python installed on your computer, it's time to start coding!

In this chapter, you will:

- Write your first Python program
- Learn what happens when you run a program with an error
- Learn how to declare a variable and inspect its value
- Learn how to write comments

Ready to begin your Python journey? Let's go!

3.1 Write a Python Program

If you don't already have IDLE open, then go ahead and open it. There are two main windows that you'll work with in IDLE: the **interactive window**, which is the one that opens when you start IDLE, and the **editor window**.

You can type code into both the interactive window and the editor window. The difference between the two windows is in how they execute code. In this section, you'll learn how to execute Python code in both windows.

The Interactive Window

IDLE's interactive window contains a **Python shell**, which is a textual user interface used to interact with the Python language. You can type a bit of Python code into the interactive window and press `Enter` to immediately see the results. Hence the name *interactive* window.

The interactive window opens automatically when you start IDLE. You'll see the following text, with some minor differences depending on your setup, displayed at the top of the window:

```
Python 3.9.1 (tags/v3.9.1:1b293b6)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This text shows the version of Python that IDLE is running. You can also see information about your operating system and some commands you can use to get help and view information about Python.

The `>>>` symbol in the last line is called the **prompt**. This is where you'll type in your code.

Go ahead and type `1 + 1` at the prompt and press `Enter`:

```
>>> 1 + 1  
2  
>>>
```

Python evaluates the expression, displays the result (2), then displays another prompt. Every time you run some code in the interactive window, a new prompt appears directly below the result.

Executing Python in the interactive window can be described as a loop with three steps:

1. Python **reads** the code entered at the prompt.
2. Python **evaluates** the code.
3. Python **prints** the result and waits for more input.

This loop is commonly referred to as a **read-evaluate-print** loop and is abbreviated as **REPL**. Python programmers sometimes refer to the Python shell as the Python REPL, or just “the REPL” for short.

Let’s try something a little more interesting than adding numbers. A rite of passage for every programmer is writing a program that prints the phrase “Hello, World” on the screen.

At the prompt in the interactive window, type the word `print` followed by a set of parentheses with the text “Hello, World” inside:

```
>>> print("Hello, World")  
Hello, World
```

A **function** is code that performs some task and can be invoked by a name. The above code invokes, or **calls**, the `print()` function with the text “Hello, World” as input.

The parentheses tell Python to call the `print()` function. They also enclose everything that gets sent to the function as input. The quotation marks indicate that "Hello, World" really is text and not something else.

Note

IDLE **highlights** parts of your code in different colors as you type to make it easier for you to identify the different parts.

By default, functions are highlighted in purple and text is highlighted in green.

The interactive window executes a single line of code at a time. This is useful for trying out small code examples and exploring the Python language, but it has a major limitation: you have to enter your code one line at a time!

Alternatively, you can save Python code in a text file and execute all of the code in the file to run an entire program.

The Editor Window

You'll write your Python files using IDLE's editor window. You can open the editor window by selecting `File > New File` from the menu at the top of the interactive window.

The interactive window stays open when you open the editor window. It displays the output generated by code in the editor window, so you'll want to arrange the two windows so that you can see them both at the same time.

In the editor window, type in the same code you used to print "Hello, World" in the interactive window:

```
print("Hello, World")
```

IDLE highlights code typed into the editor window just like in the interactive window.

Important

When you write code in a Python file, you don't need to include the >>> prompt.

Before you run your program, you need to save it. Select **File** > **Save** from the menu and save the file as `hello_world.py`.

Note

On some systems, the default directory for saving files in IDLE is the Python installation directory. **Do not** save your files to this directory. Instead, save them to your desktop or to a folder in your user's home directory.

The `.py` extension indicates that a file contains Python code. In fact, saving your file with any other extension removes the code highlighting. IDLE only highlights Python code when it's stored in a `.py` file.

Running Python Programs in the Editor Window

To run your program, select **Run** > **Run Module** from the menu in the editor window.

Note

Pressing **F5** also runs a program from the editor window.

Program output always appears in the interactive window.

Every time you run code from a file, you'll see something like the following output in the interactive window:

```
>>> ===== RESTART =====
```

IDLE restarts the Python interpreter, which is the computer program that actually executes your code, every time you run a file. This makes sure that programs are executed the same way each time.

Opening Python Files in the Editor Window

To open an existing file in IDLE, select **File > Open** from the menu, then select the file you want to open. IDLE opens every file in a new editor window, so you can have several files open at the same time.

You can also open a file from a file manager, such as Windows Explorer or macOS Finder. Right-click the file icon and select **Edit with IDLE** to open the file in IDLE's editor window.

Double-clicking on a `.py` file from a file manager executes the program. However, this usually runs the file with the system Python, and the program window disappears immediately after the program terminates—often before you can even see any output.

For now, the best way to run your Python programs is to open them in IDLE's editor window and run them from there.

3.2 Mess Things Up

Everybody makes mistakes—especially while programming! In case you haven't made any mistakes yet, let's get a head start and mess something up on purpose to see what happens.

Mistakes in programs are called **errors**. You'll experience two main types of errors: syntax errors and runtime errors.

Syntax Errors

A **syntax error** occurs when you write code that isn't allowed in the Python language.

Let's create a syntax error by removing the last quotation mark from the code in the `hello_world.py` file that you created in the last section:

```
print("Hello, World)
```

Save the file and press **F5** to run it. The code won't run! IDLE displays an alert box with the following message:

```
EOL while scanning string literal.
```

There are two terms in this message that may be unfamiliar:

1. A **string literal** is text enclosed in quotation marks. "Hello, World" is a string literal.
2. **EOL** stands for **end of line**.

So, the message tells you that Python got to the end of a line while reading a string literal. String literals must be terminated with a quotation mark before the end of a line.

IDLE highlights the line containing `print("Hello, World)` in red to help you quickly find the line of code with the syntax error. Without the second quotation mark, everything after the first quotation mark—including the closing parenthesis—is part of a string literal.

Runtime Errors

IDLE catches syntax errors before a program starts running. In contrast, **runtime errors** only occur while a program is running.

To generate a runtime error, remove both quotation marks in the `hello_world.py` file:

```
print(Hello, World)
```

Did you notice how the text color changed to black when you removed the quotation marks? IDLE no longer recognizes `Hello, World` as text. What do you think will happen when you run the program? Press **F5** to find out!

The following text displays in red in the interactive window:

```
Traceback (most recent call last):
  File "/home/hello_world.py", line 1, in <module>
    print(Hello, World)
NameError: name 'Hello' is not defined
```

Whenever an error occurs, Python stops executing the program and displays several lines of text called a **traceback**. The traceback shows useful information about the error.

Tracebacks are best read from the bottom up:

- The last line of the traceback tells you the name of the error and the error message. In this case, a `NameError` occurred because the name `Hello` is not defined anywhere.
- The second to last line shows you the code that produced the error. There's only one line of code in `hello_world.py`, so it's not hard to guess where the problem is. This information is more helpful for larger files.
- The third to last line tells you the name of the file and the line number so you can go to the exact spot in your code where the error occurred.

In the next section, you'll see how to define names for values in your code. Before you move on, though, you can get some practice with syntax errors and runtime errors by working on the review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that IDLE won't run because it has a syntax error.
2. Write a program that crashes only while it's running because it has a runtime error.

3.3 Create a Variable

In Python, **variables** are names that can be assigned a value and then used to refer to that value throughout your code.

Variables are fundamental to programming for two reasons:

1. **Variables keep values accessible:** For example, you can assign the result of some time-consuming operation to a variable so that your program doesn't have to perform the operation each time you need to use the result.
2. **Variables give values context:** The number 28 could mean lots of different things, such as the number of students in a class, the number of times a user has accessed a website, and so on. Giving the value 28 a name like `num_students` makes the meaning of the value clear.

In this section, you'll learn how to use variables in your code, as well as some of the conventions Python programmers follow when choosing names for variables.

The Assignment Operator

An **operator** is a symbol, such as `+`, that performs an operation on one or more values. For example, the `+` operator takes two numbers, one to the left of the operator and one to the right, and adds them together.

Values are assigned to variable names using a special symbol called the **assignment operator** (`=`) . The `=` operator takes the value to the right of the operator and assigns it to the name on the left.

Let's modify the `hello_world.py` file from the previous section to assign some text in a variable before printing it to the screen:

```
>>> greeting = "Hello, World"  
>>> print(greeting)  
Hello, world
```

On the first line, you create a variable named `greeting` and assign it the value "Hello, World" using the `=` operator.

`print(greeting)` displays the output `Hello, World` because Python looks for the name `greeting`, finds that it's been assigned the value "Hello, World", and replaces the variable name with its value before calling the function.

If you hadn't executed `greeting = "Hello, World"` before executing `print(greeting)`, then you would have seen a `NameError` like you did when you tried to execute `print(Hello, World)` in the previous section.

Note

Although `=` looks like the equals sign from mathematics, it has a different meaning in Python. This distinction is important and can be a source of frustration for beginner programmers.

Just remember, whenever you see the `=` operator, whatever is to the right of it is being assigned to a variable on the left.

Variable names are **case sensitive**, so a variable named `greeting` is not the same as a variable named `Greeting`. For instance, the following code produces a `NameError`:

```
>>> greeting = "Hello, World"  
>>> print(Greeting)  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
NameError: name 'Greeting' is not defined
```

If you have trouble with an example in this book, double-check that every character in your code—including spaces—matches the example *exactly*. Computers have no common sense, so being almost correct isn't good enough!

Rules for Valid Variable Names

Variable names can be as long or as short as you like, but there are a few rules that you must follow. Variable names may contain uppercase and lowercase letters (A–Z, a–z), digits (0–9), and underscores (_), but they cannot begin with a digit.

For example, each of the following is a valid Python variable name:

- string1
- _a1p4a
- list_of_names

The following aren't valid variable names because they start with a digit:

- 9lives
- 99_balloons
- 2beOrNot2Be

In addition to English letters and digits, Python variable names may contain many different valid Unicode characters.

Unicode is a standard for digitally representing characters used in most of the world's writing systems. That means variable names can contain letters from non-English alphabets, such as decorated letters like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it's a good idea to avoid them if you're going to share your code with people in different regions.

Note

You'll learn more about Unicode in chapter 12.

You can also read about Python's [support for Unicode](#) in the official Python documentation.

Just because a variable name is valid doesn't necessarily mean that it's a good name.

Choosing a good name for a variable can be surprisingly difficult. Fortunately, there are some guidelines that you can follow to help you choose better names.

Descriptive Names Are Better Than Short Names

Descriptive variable names are essential, especially for complex programs. Writing descriptive names often requires using multiple words. Don't be afraid to use long variable names.

In the following example, the value 3600 is assigned to the variable `s`:

```
s = 3600
```

The name `s` is totally ambiguous. Using a full word makes it a lot easier to understand what the code means:

```
seconds = 3600
```

`seconds` is a better name than `s` because it provides more context. But it still doesn't convey the full meaning of the code. Is 3600 the number of seconds it takes for a process to finish, or is it the length of a movie? There's no way to tell.

The following name leaves no doubt about what the code means:

```
seconds_per_hour = 3600
```

When you read the above code, there's no question that `3600` is the number of seconds in an hour. `seconds_per_hour` takes longer to type than both the single letter `s` and the word `seconds`, but the payoff in clarity is massive.

Although naming variables descriptively means using longer variable names, you should avoid using excessively long names. A good rule of thumb is to limit variable names to three or four words maximum.

Python Variable Naming Conventions

In many programming languages, it's common to write variable names in **mixedCase**. In this system, you capitalize the first letter of every word except the first and leave all other letters in lowercase. For example, `numStudents` and `listOfNames` are written in mixedCase.

In Python, however, it's more common to write variable names in **lower_case_with_underscores**. In this system, you leave every letter in lowercase and separate each word with an underscore. For instance, both `num_students` and `list_of_names` are written using the lower_case_with_underscores system.

There's no rule mandating that you write your variable names in lower_case_with_underscores. The practice is codified, though, in a document called [PEP 8](#), which is widely regarded as the official style guide for writing Python.

Note

PEP stands for **Python Enhancement Proposal**. A PEP is a design document used by the Python community to propose new features to the language.

Following the standards outlined in PEP 8 ensures that your Python code is readable by most Python programmers. This makes sharing code and collaborating with other people easier for everyone involved.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Using the interactive window, display some text using `print()`.
2. Using the interactive window, assign a string literal to a variable. Then print the contents of the variable using the `print()` function.
3. Repeat the first two exercises using the editor window.

3.4 Inspect Values in the Interactive Window

Type the following into IDLE's interactive window:

```
>>> greeting = "Hello, World"  
>>> greeting  
'Hello, World'
```

When you press `Enter` after typing `greeting` a second time, Python prints the string literal assigned to `greeting` even though you didn't use the `print()` function. This is called **variable inspection**.

Now print the string assigned to `greeting` using the `print()` function:

```
>>> print(greeting)  
Hello, World
```

Can you spot the difference between the output displayed by using `print()` and the output displayed by just entering the variable name and pressing `Enter`?

When you type the variable name `greeting` and press `Enter`, Python prints the value assigned to the variable as it appears in your code. You assigned the string literal "Hello, World" to `greeting`, which is why '`Hello, World`' is displayed with quotation marks.

Note

String literals can be created with single or double quotation marks in Python. At *Real Python*, we use double quotes wherever possible, whereas IDLE output appears in single quotes by default.

Both "Hello, World" and 'Hello, World' mean the same thing in Python—what's most important is that you be consistent in your usage. You'll learn more about strings in chapter 4.

On the other hand, `print()` displays a more human-readable representation of the variable's value which, for string literals, means displaying the text without quotation marks.

Sometimes, both printing and inspecting a variable produce the same output:

```
>>> x = 2
>>> x
2
>>> print(x)
2
```

Here, you assign the number 2 to `x`. Both using `print(x)` and inspecting `x` display output without quotation marks because 2 is a number and not text. In most cases, though, variable inspection gives you more useful information than `print()`.

Suppose you have two variables: `x`, which is assigned the number 2, and `y`, which is assigned the string literal "2". In this case, `print(x)` and `print(y)` both display the same thing:

```
>>> x = 2
>>> y = "2"
>>> print(x)
2
>>> print(y)
2
```

However, inspecting `x` and `y` shows the difference between each variable's value:

```
>>> x
2
>>> y
'2'
```

The key takeaway here is that `print()` displays a readable representation of a variable's value, while variable inspection displays the value as it appears in the code.

Keep in mind that variable inspection works only in the interactive window. For example, try running the following program from the editor window:

```
greeting = "Hello, World"
greeting
```

The program executes without any errors, but it doesn't display any output!

3.5 Leave Yourself Helpful Notes

Programmers sometimes read code they wrote a while ago and wonder, “What does this do?” When you haven’t looked at code in a while, it can be difficult to remember why you wrote it the way you did!

To help avoid this problem, you can leave comments in your code. **Comments** are lines of text that don't affect the way a program runs. They document what code does or why the programmer made certain decisions.

How to Write a Comment

The most common way to write a comment is to begin a new line in your code with the # character. When you run your code, Python ignores lines starting with #.

Comments that start on a new line are called **block comments**. You can also write **inline comments**, which are comments that appear on the same line as the code they reference. Just put a # at the end of the line of code, followed by the text in your comment.

Here's an example of a program with both kinds of comments:

```
# This is a block comment.  
greeting = "Hello, World"  
print(greeting) # This is an inline comment.
```

Of course, you can still use the # symbol inside a string. For instance, Python won't mistake the following for the start of a comment:

```
>>> print("#1")  
#1
```

In general, it's a good idea to keep comments as short as possible, but sometimes you need to write more than reasonably fits on a single line. In that case, you can continue your comment on a new line that also begins with the # symbol:

```
# This is my first program.  
# It prints the phrase "Hello, World"  
# The comments are longer than the code!  
greeting = "Hello, World"  
print(greeting)
```

You can also use comments to **comment out** code while you're testing a program. Putting a # at the beginning of a line of code lets you run your program as if that line of code didn't exist, but it doesn't actually delete the code.

To comment out a section of code in IDLE, highlight one or more lines to be commented and press:

- **Windows:** `Alt + 3`
- **macOS:** `Ctrl + 3`
- **Ubuntu Linux:** `Ctrl + D`

To remove comments, highlight the commented lines and press:

- **Windows:** `Alt + 4`
- **macOS:** `Ctrl + 4`
- **Ubuntu Linux:** `Ctrl + Shift + D`

Now let's look at some common conventions for code comments.

Conventions and Pet Peeves

According to [PEP 8](#), comments should always be written in complete sentences with a single space between the # and the first word of the comment:

```
# This comment is formatted to PEP 8.  
  
#this one isn't
```

For inline comments, PEP 8 recommends at least two spaces between the code and the # symbol:

```
phrase = "Hello, World" # This comment is PEP 8 compliant.  
print(phrase) # This comment isn't.
```

PEP 8 recommends that comments be used sparingly. A major pet peeve among programmers is comments that describe what is already obvious from reading the code.

For example, the comment in the following code is unnecessary:

```
# Print "Hello, World"  
print("Hello, World")
```

The comment is unnecessary because the code itself explicitly describes what's happening. Comments are best used to clarify code that may be difficult to understand or to explain why something is coded a certain way.

3.6 Summary and Additional Resources

In this chapter, you wrote and executed your first Python program! You wrote a small program that displays the text "Hello, World" using the `print()` function.

Then you learned about **syntax errors**, which occur *before* IDLE executes a program that contains invalid Python code, and **runtime errors**, which only occur *while* a program is running.

You saw how to assign values to **variables** using the **assignment operator** (`=`) and how to inspect variables in the interactive window.

Finally, you learned how to write helpful **comments** in your code for when you or someone else looks at it in the future.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-first-program

Additional Resources

To learn more, check out the following resources:

- “[11 Beginner Tips for Learning Python Programming](#)”
- “[Writing Comments in Python \(Guide\)](#)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 4

Strings and String Methods

Many programmers, regardless of their specialty, deal with text on a daily basis. For example, web developers work with text input from web forms. Data scientists process text to extract data and perform tasks like sentiment analysis, which can help identify and classify opinions in a body of text.

Collections of text in Python are called **strings**. Special functions called **string methods** are used to manipulate strings. There are string methods for changing a string from lowercase to uppercase, removing whitespace from the beginning or end of a string, replacing parts of a string with different text, and much more.

In this chapter, you'll learn how to:

- Manipulate strings with string methods
- Work with user input
- Deal with strings of numbers
- Format strings for printing

Let's get started!

4.1 What Is a String?

In chapter 3, you created the string "Hello, World" and printed it in IDLE's interactive window using `print()`. In this section, you'll get a deeper look into exactly what strings are and the various ways you can create them in Python.

The String Data Type

Strings are one of the fundamental Python data types. The term **data type** refers to what kind of data a value represents. Strings are used to represent text.

Note

There are several other data types built into Python. For example, you'll learn about numerical data types in chapter 5 and Boolean data types in chapter 8.

We say that strings are a **fundamental data type** because they can't be broken down into smaller values of a different type. Not all data types are fundamental. You'll learn about compound data types, also known as **data structures**, in chapter 9.

The string data type has a special abbreviated name in Python: `str`. You can see this by using `type()`, which is a function used to determine the data type of a given value.

Type the following into IDLE's interactive window:

```
>>> type("Hello, World")
<class 'str'>
```

The output `<class 'str'>` indicates that the value "Hello, World" is an instance of the `str` data type. That is, "Hello, World" is a string.

Note

For now, you can think of the word *class* as a synonym for *data type*, although it actually refers to something more specific. You'll see just what a class is in chapter 10.

`type()` also works for values that have been assigned to a variable:

```
>>> phrase = "Hello, World"  
>>> type(phrase)  
<class 'str'>
```

Strings have three important properties:

1. Strings contain individual letters or symbols called **characters**.
2. Strings have a **length**, defined as the number of characters the string contains.
3. Characters in a string appear in a **sequence**, which means that each character has a numbered position in the string.

Let's take a closer look at how strings are created.

String Literals

As you've already seen, you can create a string by surrounding some text with quotation marks:

```
string1 = 'Hello, World'  
string2 = "1234"
```

You can use either single quotes (`string1`) or double quotes (`string2`) to create a string as long as you use the same type at the beginning and end of the string.

Whenever you create a string by surrounding text with quotation marks, the string is called a **string literal**. The name indicates that the string is literally written out in your code. All the strings you've seen thus far are string literals.

Note

Not every string is a string literal. Sometimes strings are input by a user or read from a file. Since they're not typed out with quotation marks in your code, they're not string literals.

The quotes surrounding a string are called **delimiters** because they tell Python where a string begins and where it ends. When one type of quotes is used as the delimiter, the other type can be used inside the string:

```
string3 = "We're #1!"  
string4 = 'I said, "Put it over by the llama."'
```

After Python reads the first delimiter, it considers all the characters after it part of the string until it reaches a second matching delimiter. This is why you can use a single quote in a string delimited by double quotes, and vice versa.

If you try to use double quotes inside a string delimited by double quotes, you'll get an error:

```
>>> text = "She said, "What time is it?""  
      File "<stdin>", line 1  
      text = "She said, "What time is it?""  
                           ^  
SyntaxError: invalid syntax
```

Python throws a `SyntaxError` because it thinks the string ends after the second `",`, and it doesn't know how to interpret the rest of the line. If you need to include a quotation mark that matches the delimiter inside a string, then you can **escape** the character using a backslash:

```
>>> text = "She said, \"What time is it?\""  
>>> print(text)  
She said, "What time is it?"
```

Note

When you work on a project, it's a good idea to use only single quotes or only double quotes to delimit every string.

Keep in mind that there really isn't a right or wrong choice! The goal is to be consistent because consistency helps make your code easier to read and understand.

Strings can contain any valid Unicode character. For example, the string "We're #1!" contains the pound sign (#) and "1234" contains numbers. "xPýþhøøx" is also a valid Python string!

Determine the Length of a String

The number of characters contained in a string, including spaces, is called the **length** of the string. For example, the string "abc" has a length of 3, and the string "Don't Panic" has a length of 11.

Python has a built-in `len()` function that you can use to determine the length of a string. To see how it works, type the following into IDLE's interactive window:

```
>>> len("abc")
3
```

You can also use `len()` to get the length of a string that's assigned to a variable:

```
>>> letters = "abc"
>>> len(letters)
3
```

First, you assign the string "abc" to the variable `letters`. Then you use `len()` to get the length of `letters`, which is 3.

Multiline Strings

The [PEP 8](#) style guide recommends that each line of Python code contain no more than seventy-nine characters—including spaces.

Note

PEP 8's seventy-nine-character line length is a recommendation, not a rule. Some Python programmers prefer a slightly longer line length.

In this book, we'll strictly follow PEP 8's recommended line length.

Whether you follow PEP 8 or choose a longer line length, sometimes you'll need to create string literals with more characters than your chosen limit.

To deal with long strings, you can break them up across multiple lines into **multiline strings**. For example, suppose you need to fit the following text into a string literal:

This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn't the small green pieces of paper that were unhappy.

— Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

This paragraph contains far more than seventy-nine characters, so any line of code containing the paragraph as a string literal violates PEP 8. So, what do you do?

There are a couple of ways to tackle this. One way is to break the string up across multiple lines and put a backslash (\) at the end of all but the

last line. To be PEP 8 compliant, the total length of the line, including the backslashes, must be seventy-nine characters or fewer.

Here's how you could write the paragraph as a multiline string using the backslash method:

```
paragraph = "This planet has—or rather had—a problem, which was \
this: most of the people living on it were unhappy for pretty much \
of the time. Many solutions were suggested for this problem, but \
most of these were largely concerned with the movements of small \
green pieces of paper, which is odd because on the whole it wasn't \
the small green pieces of paper that were unhappy."
```

Notice that you don't have to close each line with a quotation mark. Normally, Python would get to the end of the first line and complain that you didn't close the string with a matching double quote. With a backslash at the end, you can keep writing the same string on the next line.

When you `print()` a multiline string that's broken up by backslashes, the output is displayed on a single line:

```
>>> long_string = "This multiline string is \
displayed on one line"
>>> print(long_string)
This multiline string is displayed on one line
```

You can also create multiline strings using triple quotes ("""" or ''') as delimiters. Here's how to write a long paragraph using this approach:

```
paragraph = """This planet has—or rather had—a problem, which was
this: most of the people living on it were unhappy for pretty much
of the time. Many solutions were suggested for this problem, but
most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't
the small green pieces of paper that were unhappy."""
```

Triple-quoted strings preserve whitespace, including newlines. This means that running `print(paragraph)` would display the string on multiple lines, just as it appears in the string literal. This may or may not be what you want, so you'll need to think about the desired output before you choose how to write a multiline string.

To see how whitespace is preserved in a triple-quoted string, type the following into IDLE's interactive window:

```
>>> print("""An example of a
...     string that spans across multiple lines
...         and also preserves whitespace.""")
An example of a
    string that spans across multiple lines
        and also preserves whitespace.
```

Notice how the second and third lines in the output are indented in exactly the same way as the string literal.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Print a string that uses double quotation marks inside the string.
2. Print a string that uses an apostrophe inside the string.
3. Print a string that spans multiple lines with whitespace preserved.
4. Print a string that is coded on multiple lines but gets printed on a single line.

4.2 Concatenation, Indexing, and Slicing

Now that you know what a string is and how to declare string literals in your code, let's explore some of the things you can do with strings.

In this section, you'll learn about three basic string operations:

1. **Concatenation**, which joins two strings together
2. **Indexing**, which gets a single character from a string
3. **Slicing**, which gets several characters from a string at once

Let's dive in!

String Concatenation

You can combine, or **concatenate**, two strings using the + operator:

```
>>> string1 = "abra"  
>>> string2 = "cadabra"  
>>> magic_string = string1 + string2  
>>> magic_string  
'abracadabra'
```

In this example, the string concatenation occurs on the third line. You concatenate `string1` and `string2` using +, and then you assign the result to the variable `magic_string`. Notice that the two strings are joined without any whitespace between them.

You can use string concatenation to join two related strings, such as joining a first name and a last name into a full name:

```
>>> first_name = "Arthur"  
>>> last_name = "Dent"  
>>> full_name = first_name + " " + last_name  
>>> full_name  
'Arthur Dent'
```

Here, you use string concatenation twice on the same line. First, you concatenate `first_name` with " " to ensure a space appears after the first name in the final string. This produces the string "Arthur ", which you then concatenate with `last_name` to produce the full name "Arthur Dent".

String Indexing

Each character in a string has a numbered position called an **index**. You can access the character at the n th position by putting the number n between two square brackets (`[]`) immediately after the string:

```
>>> flavor = "fig pie"  
>>> flavor[1]  
'i'
```

`flavor[1]` returns the character at position 1 in `"fig pie"`, which is `i`.

Wait. Isn't `f` the first character of `"fig pie"`?

In Python—and in most other programming languages—counting always starts at zero. To get the character at the beginning of a string, you need to access the character at position 0:

```
>>> flavor[0]  
'f'
```

Important

Forgetting that counting starts with zero and trying to access the first character in a string with the index 1 results in an **off-by-one error**.

Off-by-one errors are a common source of frustration for beginning and experienced programmers alike!

The following figure shows the index for each character of the string `"fig pie"`:

	f		i		g				p		i		e	
0		1		2		3		4		5		6		

If you try to access an index beyond the end of a string, then Python raises an `IndexError`:

```
>>> flavor[9]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    flavor[9]
IndexError: string index out of range
```

The largest index in a string is always one less than the string's length. Since "fig pie" has a length of seven, the largest index allowed is 6.

Strings also support negative indices:

```
>>> flavor[-1]
'e'
```

The last character in a string has index -1, which for "fig pie" is the letter e. The second to last character i has index -2, and so on.

The following figure shows the negative index for each character in the string "fig pie":

	f		i		g				p		i		e	
-7		-6		-5		-4		-3		-2		-1		

Just like with positive indices, Python raises an `IndexError` if you try to access a negative index less than the index of the first character in the string:

```
>>> flavor[-10]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    flavor[-10]
IndexError: string index out of range
```

Negative indices may not seem useful at first, but sometimes they're a better choice than a positive index.

For example, suppose a string input by a user is assigned to the variable `user_input`. If you need to get the last character of the string, how do you know what index to use?

One way to get the last character of a string is to calculate the final index using `len()`:

```
final_index = len(user_input) - 1  
last_character = user_input[final_index]
```

Getting the final character with the index `-1` takes less typing and doesn't require an intermediate step to calculate the final index:

```
last_character = user_input[-1]
```

String Slicing

Suppose you need a string containing just the first three letters of the string "fig pie". You could access each character by index and concatenate them like this:

```
>>> first_three_letters = flavor[0] + flavor[1] + flavor[2]  
>>> first_three_letters  
'fig'
```

If you need more than just the first few letters of a string, then getting each character individually and concatenating them together is clumsy and long-winded. Fortunately, Python provides a way to do this with much less typing.

You can extract a portion of a string, called a **substring**, by inserting a colon between two index numbers set inside square brackets like this:

```
>>> flavor = "fig pie"  
>>> flavor[0:3]  
'fig'
```

`flavor[0:3]` returns the first three characters of the string assigned to `flavor`, starting with the character at index 0 and going up to but not including the character at index 3. The `[0:3]` part of `flavor[0:3]` is called a **slice**. In this case, it returns a slice of "fig pie". Yum!

String slices can be confusing because the substring returned by the slice includes the character whose index is the first number but doesn't include the character whose index is the second number.

To remember how slicing works, you can think of a string as a sequence of square slots. The left and right boundaries of each slot are numbered sequentially from zero up to the length of the string, and each slot is filled with a character in the string.

Here's what this looks like for the string "fig pie":

	f		i		g				p		i		e	
0	1	2	3	4	5	6	7							

So, for "fig pie", the slice `[0:3]` returns the string "fig", and the slice `[3:7]` returns the string " pie".

If you omit the first index in a slice, then Python assumes you want to start at index 0:

```
>>> flavor[:3]
'fig'
```

The slice `[:3]` is equivalent to the slice `[0:3]`, so `flavor[:3]` returns the first three characters in the string "fig pie".

Similarly, if you omit the second index in the slice, then Python assumes you want to return the substring that begins with the character

whose index is the first number in the slice and ends with the last character in the string:

```
>>> flavor[3:]  
' pie'
```

For "fig pie", the slice [3:] is equivalent to the slice [3:7]. Since the character at index 3 is a space, flavor[3:7] returns the substring that starts with the space and ends with the last letter: " pie".

If you omit both the first and second numbers in a slice, you get a string that starts with the character at index 0 and ends with the last character. In other words, omitting both numbers in a slice returns the entire string:

```
>>> flavor[:]  
'fig pie'
```

It's important to note that, unlike with string indexing, Python won't raise an `IndexError` when you try to slice between boundaries that fall outside the beginning or ending boundaries of a string:

```
>>> flavor[:14]  
'fig pie'  
>>> flavor[13:15]  
''
```

In this example, the first line gets the slice from the beginning of the string up to but not including the fourteenth character. The string assigned to `flavor` has a length of seven, so you might expect Python to throw an error. Instead, it ignores any nonexistent indices and returns the entire string "fig pie".

The third line shows what happens when you try to get a slice in which the entire range is out of bounds. `flavor[13:15]` attempts to get the thirteenth and fourteenth characters, which don't exist. Instead of raising an error, Python returns the **empty string** ("").

Note

The empty string is called empty because it doesn't contain any characters. You can create it by writing two quotation marks with nothing between them:

```
empty_string = ""
```

A string with anything in it—even a space—is not empty. All the following strings are non-empty:

```
non_empty_string1 = " "
non_empty_string2 = "      "
non_empty_string3 = "           "
```

Even though these strings don't contain any *visible* characters, they are non-empty because they do contain spaces.

You can use negative numbers in slices. The rules for slices with negative numbers are exactly the same as the rules for slices with positive numbers. It helps to visualize the string as slots with the boundaries labeled with negative numbers:

	f		i		g				p		i		e	
-7	-6	-5	-4	-3	-2	-1								

Just like before, the slice `[x:y]` returns the substring starting at index `x` and going up to but not including `y`. For instance, the slice `[-7:-4]` returns the first three letters of the string "fig pie":

```
>>> flavor[-7:-4]
'fig'
```

Notice, however, that the rightmost boundary of the string does not have a negative index. The logical choice for that boundary would seem to be the number 0, but that doesn't work.

Instead of returning the entire string, `[-7:0]` returns the empty string:

```
>>> flavor[-7:0]
''
```

This happens because the second number in a slice must correspond to a boundary that is to the right of the boundary corresponding to the first number, but both `-7` and `0` correspond to the leftmost boundary in the figure.

If you need to include the final character of a string in your slice, then you can omit the second number:

```
>>> flavor[-7:]
'fig pie'
```

Of course, using `flavor[-7:]` to get the entire string is a bit odd considering that you can use the variable `flavor` without the slice to get the same result!

Slices with negative indices are useful, though, for getting the last few characters in a string. For example, `flavor[-3:]` is "pie".

Strings Are Immutable

To wrap this section up, let's discuss an important property of string objects. Strings are **immutable**, which means that you can't change them once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
>>> word = "goal"
>>> word[0] = "f"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    word[0] = "f"
TypeError: 'str' object does not support item assignment
```

Python throws a `TypeError` and tells you that `str` objects don't support item assignment.

If you want to alter a string, then you must create an entirely new string. To change the string "goal" to the string "foal", you can use a string slice to concatenate the letter "f" with everything but the first letter of the word "goal":

```
>>> word = "goal"  
>>> word = "f" + word[1:]  
>>> word  
'foal'
```

First, you assign the string "goal" to the variable `word`. Then you concatenate the slice `word[1:]`, which is the string "oal", with the letter "f" to get the string "foal". If you're getting a different result here, then make sure you're including the colon character (:) as part of the string slice.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a string and print its length using `len()`.
2. Create two strings, concatenate them, and print the resulting string.
3. Create two strings, use concatenation to add a space between them, and print the result.
4. Print the string "zing" by using slice notation to specify the correct range of characters in the string "bazinga".

4.3 Manipulate Strings With Methods

Strings come bundled with special functions called **string methods** that you can use to work with and manipulate strings. There are numerous string methods available, but we'll focus on some of the most commonly used ones.

In this section, you'll learn how to:

- Convert a string to uppercase or lowercase
- Remove whitespace from a string
- Determine if a string begins or ends with certain characters

Let's go!

Converting String Case

To convert a string to all lowercase letters, you use the string's `.lower()` method. This is done by tacking `.lower()` onto the end of the string itself:

```
>>> "Jean-Luc Picard".lower()  
'jean-luc picard'
```

The dot (.) tells Python that what follows is the name of a method—the `lower()` method in this case.

Note

We'll refer to string methods with a dot (.) at the beginning of their names. For example, `.lower()` is written with a leading dot instead of as `lower()`.

This makes it easier to differentiate functions that are string methods from built-in functions like `print()` and `type()`.

String methods don't just work on string literals. You can also use `.lower()` on a string assigned to a variable:

```
>>> name = "Jean-Luc Picard"
>>> name.lower()
'jean-luc picard'
```

The opposite of `.lower()` is `.upper()`, which converts every character in a string to uppercase:

```
>>> name.upper()
'JEAN-LUC PICARD'
```

Compare the `.upper()` and `.lower()` string methods to the `len()` function you saw in the last section. Aside from the different results of these functions, the important distinction here is how they're used.

`len()` is a stand-alone function. If you want to determine the length of the `name` string, then you call the `len()` function directly:

```
>>> len(name)
15
```

On the other hand, `.upper()` and `.lower()` must be used in conjunction with a string. They do not exist independently.

Removing Whitespace From a String

Whitespace is any character that is printed as blank space. This includes things like spaces and **line feeds**, which are special characters that move output to a new line.

Sometimes you need to remove whitespace from the beginning or end of a string. This is especially useful when working with strings that come from user input, which may include extra whitespace characters by accident.

There are three string methods that you can use to remove whitespace from a string:

1. `.rstrip()`

2. `.lstrip()`

3. `.strip()`

`.rstrip()` removes whitespace from the right side of a string:

```
>>> name = "Jean-Luc Picard      "
>>> name
'Jean-Luc Picard      '
>>> name.rstrip()
'Jean-Luc Picard'
```

In this example, the string "Jean-Luc Picard " has five trailing spaces. You use `.rstrip()` to remove trailing spaces from the right-hand side of the string. This returns the new string "Jean-Luc Picard", which no longer has the spaces at the end.

`.lstrip()` works just like `.rstrip()`, except that it removes whitespace from the left-hand side of the string:

```
>>> name = "      Jean-Luc Picard"
>>> name
'      Jean-Luc Picard'
>>> name.lstrip()
'Jean-Luc Picard'
```

To remove whitespace from both the left and the right sides of the string at the same time, use `.strip()`:

```
>>> name = "      Jean-Luc Picard      "
>>> name
'      Jean-Luc Picard      '
>>> name.strip()
'Jean-Luc Picard'
```

It's important to note that none of `.rstrip()`, `.lstrip()`, or `.strip()` removes whitespace from the middle of the string. In each of the previous examples, the space between "Jean-Luc" and "Picard" is preserved.

Determine If a String Starts or Ends With a Particular String

When you work with text, sometimes you need to determine if a given string starts with or ends with certain characters. You can use two string methods to solve this problem: `.startswith()` and `.endswith()`.

Let's look at an example. Consider the string "Enterprise". Here's how you use `.startswith()` to determine if the string starts with the letters e and n:

```
>>> starship = "Enterprise"
>>> starship.startswith("en")
False
```

You tell `.startswith()` which characters to search for by providing a string containing those characters. So, to determine if "Enterprise" starts with the letters e and n, you call `.startswith("en")`. This returns False. Why do you think that is?

If you guessed that `.startswith("en")` returns False because "Enterprise" starts with a capital E, then you're absolutely right! The `.startswith()` method is **case sensitive**. To get `.startswith()` to return True, you need to provide it with the string "En":

```
>>> starship.startswith("En")
True
```

You can use `.endswith()` to determine if a string ends with certain characters:

```
>>> starship.endswith("rise")
True
```

Just like `.startswith()`, the `.endswith()` method is case sensitive:

```
>>> starship.endswith("risE")
False
```

Note

The `True` and `False` values are not strings. They are a special kind of data type called a **Boolean value**. You'll learn more about Boolean values in chapter 8.

String Methods and Immutability

Recall from the previous section that strings are immutable—they can't be changed once they've been created. Most string methods that alter a string, like `.upper()` and `.lower()`, actually return copies of the original string with the appropriate modifications.

If you aren't careful, this can introduce subtle bugs into your program. Try this out in IDLE's interactive window:

```
>>> name = "Picard"
>>> name.upper()
'PICARD'
>>> name
'Picard'
```

When you call `name.upper()`, nothing about `name` actually changes. If you need to keep the result, then you need to assign it to a variable:

```
>>> name = "Picard"
>>> name = name.upper()
>>> name
'PICARD'
```

`name.upper()` returns a new string "PICARD", which is reassigned to the `name` variable. This **overrides** the original string "Picard" that you first assigned to `name`.

Use IDLE to Discover Additional String Methods

Strings have lots of methods associated with them, and the methods introduced in this section barely scratch the surface. IDLE can help you find new string methods. To see how, first assign a string literal to a variable in the interactive window:

```
>>> starship = "Enterprise"
```

Next, type `starship` followed by a period, but do not hit `Enter`. You should see the following in the interactive window:

```
>>> starship.
```

Now wait for a couple of seconds. IDLE displays a list of every string method, which you can scroll through using the arrow keys.

A related shortcut in IDLE is the ability to use `Tab` to automatically fill in text without having to type long names. For instance, if you type only `starship.u` and hit `Tab`, then IDLE automatically fills in `starship.upper` because only one method that begins with a `u` belongs to `starship`.

This even works with variable names. Try typing just the first few letters of `starship` and pressing `Tab`. If you haven't defined any other names that share those first letters, then IDLE completes the name `starship` for you.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that converts the following strings to lowercase: `"Animals", "Badger", "Honey Bee", "Honey Badger"`. Print each lowercase string on a separate line.
2. Repeat exercise 1, but convert each string to uppercase instead of lowercase.

3. Write a program that removes whitespace from the following strings, then print out the strings with the whitespace removed:

```
string1 = "    Filet Mignon"
string2 = "Brisket      "
string3 = "  Cheeseburger  "
```

4. Write a program that prints out the result of `.startswith("be")` on each of the following strings:

```
string1 = "Becomes"
string2 = "becomes"
string3 = "BEAR"
string4 = " bEautiful"
```

5. Using the same four strings from exercise 4, write a program that uses string methods to alter each string so that `.startswith("be")` returns True for all of them.

4.4 Interact With User Input

Now that you've seen how to work with string methods, let's make things interactive!

In this section, you'll learn how to get some input from a user with `input()`. You'll write a program that asks a user to input some text and then displays that text back to them in uppercase.

Enter the following into IDLE's interactive window:

```
>>> input()
```

When you press `Enter`, it looks like nothing happens. The cursor moves to a new line, but a new `>>>` doesn't appear. Python is waiting for you to enter something!

Go ahead and type some text and press `Enter`:

```
>>> input()
Hello there!
'Hello there!'
>>>
```

The text you entered is repeated on a new line with single quotes. That's because `input()` returns as a string any text entered by the user.

To make `input()` a bit more user-friendly, you can give it a **prompt** to display to the user. The prompt is just a string that you put between the parentheses of `input()`. It can be anything you want: a word, a symbol, a phrase—anything that is a valid Python string.

`input()` displays the prompt and waits for the user to type something. When the user hits `Enter`, `input()` returns their input as a string that can be assigned to a variable and used to do something in your program.

To see how `input()` works, type the following code into IDLE's editor window:

```
prompt = "Hey, what's up? "
user_input = input(prompt)
print("You said: " + user_input)
```

Press `F5` to run the program. The text `Hey, what's up?` displays in the interactive window with a blinking cursor.

The single space at the end of the string `"Hey, what's up? "` makes sure that when the user starts to type, the text is separated from the prompt with a space. When the user types a response and presses `Enter`, their response is assigned to the `user_input` variable.

Here's a sample run of the program:

```
Hey, what's up? Mind your own business.
```

```
You said: Mind your own business.
```

Once you have input from a user, you can do something with it. For example, the following program takes user input, converts it to uppercase with `.upper()`, and prints the result:

```
response = input("What should I shout? ")
shouted_response = response.upper()
print("Well, if you insist..." + shouted_response)
```

Try typing this program into IDLE's editor window and running it. What else can you think of to do with the input?

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that takes input from the user and displays that input back.
2. Write a program that takes input from the user and displays the input in lowercase.
3. Write a program that takes input from the user and displays the number of characters in the input.

4.5 Challenge: Pick Apart Your User's Input

Write a program named `first_letter.py` that prompts the user for input with the string "Tell me your password:". The program should then determine the first letter of the user's input, convert that letter to uppercase, and display it back.

For example, if the user input is "no", then the program should display the following output:

```
The first letter you entered was: N
```

For now, it's okay if your program crashes when the user enters nothing as input—that is, when they just hit `Enter` instead of typing something. You'll learn a couple of ways to deal with this situation in an upcoming chapter.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

4.6 Working With Strings and Numbers

When you get user input using `input()`, the result is always a string. There are many other situations in which input is given to a program as a string. Sometimes those strings contain numbers that need to be fed into calculations.

In this section, you'll learn how to deal with strings of numbers. You'll see how arithmetic operations work on strings and how they often lead to surprising results. You'll also learn how to convert between strings and number types.

Using Strings With Arithmetic Operators

You've seen that string objects can hold many types of characters, including numbers. However, don't confuse numerals in a string with

actual numbers. For instance, try this bit of code out in IDLE's interactive window:

```
>>> num = "2"  
>>> num + num  
'22'
```

The `+` operator concatenates two strings together, which is why the result of `"2" + "2"` is `"22"` and not `"4"`.

You can multiply strings by a number as long as that number is an integer or whole number. Type the following into the interactive window:

```
>>> num = "12"  
>>> num * 3  
'121212'
```

`num * 3` concatenates three instances of the string `"12"` and returns the string `"121212"`.

Compare this operation to arithmetic with numbers. When you multiply the number `12` by the number `3`, the result is the same as adding three `12`s together. The same is true for a string. That is, `"12" * 3` can be interpreted as `"12" + "12" + "12"`. In general, multiplying a string by an integer n concatenates n copies of that string.

You can move the number on the right-hand side of the expression `num * 3` to the left, and the result is unchanged:

```
>>> 3 * num  
'121212'
```

What do you think happens if you use the `*` operator between two strings?

Type "12" * "3" in the interactive window and press **Enter**:

```
>>> "12" * "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Python raises a `TypeError` and tells you that you can't multiply a sequence by a non-integer.

Note

A **sequence** is any Python object that supports accessing elements by index. Strings are sequences. You'll learn about other sequence types in chapter 9.

When you use the `*` operator with a string, Python always expects an integer on the other side of the operator.

What do you think happens when you try to add a string and a number?

```
>>> "3" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Python throws a `TypeError` because it expects the objects on both sides of the `+` operator to be of the same type.

If an object on either side of `+` is a string, then Python tries to perform string concatenation. It will only perform addition if both objects are numbers. So, to add `"3" + 3` and get 6, you must first convert the string `"3"` to a number.

Converting Strings to Numbers

The `TypeError` examples in the previous section highlight a common problem when applying user input to an operation that requires a number and not a string: type mismatches.

Let's look at an example. Save and run the following program:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

If you entered the number 2 at the prompt, then you would expect the output to be 4. But in this case, you would get 22. Remember, `input()` always returns a string, so if you input 2, then `num` is assigned the string "2", not the integer 2. Therefore, the expression `num * 2` returns the string "2" concatenated with itself, which is "22".

To perform arithmetic on numbers contained in a string, you must first convert them from a string type to a number type. There are two functions that you can use to do this: `int()` and `float()`.

`int()` stands for **integer** and converts objects into whole numbers, whereas `float()` stands for **floating-point number** and converts objects into numbers with decimal points. Here's what using each one looks like in the interactive window:

```
>>> int("12")
12

>>> float("12")
12.0
```

Notice how `float()` adds a decimal point to the number. Floating-point numbers always have at least one decimal place of precision. For this reason, you can't change a string that looks like a floating-point number into an integer because you would lose everything after the decimal point.

Try converting the string "12.0" to an integer:

```
>>> int("12.0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.0'
```

Even though the extra 0 after the decimal place doesn't add any value to the number, Python won't change 12.0 into 12 because it would result in a loss of precision.

Let's revisit the program from the beginning of this section and see how to fix it. Here's the code again:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

The issue is on the line `doubled_num = num * 2` because `num` is a string and 2 is an integer.

You can fix the problem by passing `num` to either `int()` or `float()`. Since the prompts asks the user to input a number, and not specifically an integer, let's convert `num` to a floating-point number:

```
num = input("Enter a number to be doubled: ")
doubled_num = float(num) * 2
print(doubled_num)
```

Now when you run this program and input 2, you get 4.0 as expected. Try it out!

Converting Numbers to Strings

Sometimes you need to convert a number to a string. You might do this, for example, if you need to build a string from some preexisting variables that are assigned to numeric values.

As you've already seen, concatenating a number with a string produces a `TypeError`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + num_pancakes + " pancakes."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Since `num_pancakes` is a number, Python can't concatenate it with the string "I'm going to eat". To build the string, you need to convert `num_pancakes` to a string using `str()`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + str(num_pancakes) + " pancakes."
'I am going to eat 10 pancakes.'
```

You can also call `str()` on a number literal:

```
>>> "I am going to eat " + str(10) + " pancakes."
'I am going to eat 10 pancakes.'
```

`str()` can even handle arithmetic expressions:

```
>>> total_pancakes = 10
>>> pancakes_eaten = 5
>>> "Only " + str(total_pancakes - pancakes_eaten) + " pancakes left."
'Only 5 pancakes left.'
```

In the next section, you'll learn how to format strings neatly to display values in a nice, readable manner. Before you move on, though, check your understanding with the following review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a string containing an integer, then convert that string into an actual integer object using `int()`. Test that your new object is a number by multiplying it by another number and displaying the result.
2. Repeat the previous exercise, but use a floating-point number and `float()`.
3. Create a string object and an integer object, then display them side by side with a single print function call using `str()`.
4. Write a program that uses `input()` twice to get two numbers from the user, multiplies the numbers together, and displays the result. If the user enters 2 and 4, then your program should print the following text:

```
The product of 2 and 4 is 8.0.
```

4.7 Streamline Your Prints

Suppose you have a string, `name = "Zaphod"`, and two integers, `heads = 2` and `arms = 3`. You want to display them in the string `"Zaphod has 2 heads and 3 arms"`. This is called **string interpolation**, which is just a fancy way of saying that you want to insert some variables into specific locations in a string.

One way to do this is with string concatenation:

```
>>> name + " has " + str(heads) + " heads and " + str(arms) + " arms"  
'Zaphod has 2 heads and 3 arms'
```

This code isn't the prettiest, and keeping track of what goes inside or outside the quotes can be tough. Fortunately, there's another way of interpolating strings: **formatted string literals**, more commonly known as **f-strings**.

The easiest way to understand f-strings is to see them in action. Here's what the above string looks like when written as an f-string:

```
>>> f"{name} has {heads} heads and {arms} arms"  
'Zaphod has 2 heads and 3 arms'
```

There are two important things to notice about the above example:

1. The string literal starts with the letter `f` before the opening quotation mark.
2. Variable names surrounded by curly braces `{}` are replaced by their corresponding values without using `str()`.

You can also insert Python expressions between the curly braces. The expressions are replaced with their result in the string:

```
>>> n = 3  
>>> m = 4  
>>> f"{n} times {m} is {n*m}"  
'3 times 4 is 12'
```

It's a good idea to keep any expressions used in an f-string as simple as possible. Packing a bunch of complicated expressions into a string literal can result in code that is difficult to read and difficult to maintain.

f-strings are available only in Python version 3.6 and above. In earlier versions of Python, you can use `.format()` to get the same results. Returning to the Zaphod example, you can use `.format()` to format the string like this:

```
>>> "{} has {} heads and {} arms".format(name, heads, arms)  
'Zaphod has 2 heads and 3 arms'
```

f-strings are shorter and sometimes more readable than using `.format()`. You'll see f-strings used throughout this book.

For an in-depth guide to f-strings and comparisons to other string formatting techniques, check out *Real Python*'s “[Python 3's f-Strings: An Improved String Formatting Syntax \(Guide\)](#).”

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a `float` object named `weight` with the value `0.2`, and create a string object named `animal` with the value `"newt"`. Then use these objects to print the following string using only string concatenation:

```
0.2 kg is the weight of the newt.
```

2. Display the same string by using `.format()` and empty `{}` placeholders.
3. Display the same string using an f-string.

4.8 Find a String in a String

One of the most useful string methods is `.find()`. As its name implies, this method allows you to find the location of one string in another string—commonly referred to as a **substring**.

To use `.find()`, tack it to the end of a variable or a string literal with the string you want to find typed between the parentheses:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("surprise")
4
```

The value that `.find()` returns is the index of the first occurrence of the string you pass to it. In this case, “surprise” starts at the fifth character

of the string "the surprise is in here somewhere", which has index 4 because counting starts at zero.

If `.find()` doesn't find the desired substring, it will return `-1` instead:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("eyjafjallajökull")
-1
```

Keep in mind that this matching is done exactly, character by character, and is case sensitive. For example, if you try to find "SURPRISE", then `.find()` returns `-1`:

```
>>> "the surprise is in here somewhere".find("SURPRISE")
-1
```

If a substring appears more than once in a string, then `.find()` returns the index of only the first appearance, starting from the beginning of the string:

```
>>> "I put a string in your string".find("string")
8
```

There are two instances of "string" in "I put a string in your string". The first is at index 8, and the second is at index 23, but `.find()` returns only 8.

`.find()` will only accept a string as its input. If you want to find an integer in a string, then you need to pass the integer to `.find()` as a string. Otherwise, Python raises a `TypeError`:

```
>>> "My number is 555-555-5555".find(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int

>>> "My number is 555-555-5555".find("5")
13
```

Sometimes you need to find all occurrences of a particular substring and replace them with a different string. Since `.find()` returns the index of the first occurrence of a substring, you can't easily use it to perform this operation. But strings have a `.replace()` method that replaces each instance of a substring with another string.

Just like with `.find()`, you tack `.replace()` onto the end of a variable or string literal. In this case, though, you need to put two strings inside the parentheses in `.replace()` and separate them with a comma. The first string is the substring to find, and the second string is the string with which to replace each occurrence of the substring.

For example, the following code replaces each occurrence of "the truth" in the string "I'm telling you the truth; nothing but the truth" with the string "lies":

```
>>> my_story = "I'm telling you the truth; nothing but the truth!"  
>>> my_story.replace("the truth", "lies")  
"I'm telling you lies; nothing but lies!"
```

Since strings are immutable objects, `.replace()` doesn't alter `my_story`. If you immediately type `my_story` into the interactive window after running the above example, then you'll see the original string, unaltered:

```
>>> my_story  
"I'm telling you the truth; nothing but the truth!"
```

To change the value of `my_story`, you need to reassign it with the new value returned by `.replace()`:

```
>>> my_story = my_story.replace("the truth", "lies")  
>>> my_story  
"I'm telling you lies; nothing but lies!"
```

`.replace()` replaces every instance of the substring with the replacement text. If you want to replace multiple substrings in a string, then you need to use `.replace()` multiple times:

```
>>> text = "some of the stuff"
>>> new_text = text.replace("some of", "all")
>>> new_text = new_text.replace("stuff", "things")
>>> new_text
'all the things'
```

You'll have some fun with `.replace()` in the challenge in the next section.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. In one line of code, display the result of trying to `.find()` the substring "a" in the string "AAA". The result should be -1.
2. Replace every occurrence of the character "s" with "x" in the string "Somebody said something to Samantha."
3. Write a program that accepts user input with `input()` and displays the result of trying to `.find()` a particular letter in that input.

4.9 **Challenge: Turn Your User Into a L33t H4xor**

Write a program called `translate.py` that asks the user for some input with the following prompt:

```
Enter some text:
```

Use `.replace()` to convert the text entered by the user into leetspeak by making the following changes to lowercase letters:

- The letter a becomes 4
- The letter b becomes 8
- The letter e becomes 3

- The letter l becomes 1
- The letter o becomes 0
- The letter s becomes 5
- The letter t becomes 7

Your program should then display the resulting string as output. Below is a sample run of the program:

```
Enter some text: I like to eat eggs and spam.  
I 1ik3 70 347 3gg5 4nd 5p4m.
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

4.10 Summary and Additional Resources

In this chapter, you learned the ins and outs of Python string objects. You learned how to access different characters in a string using indices and slices as well as how to determine the length of a string with `len()`.

Strings come with numerous methods. The `.upper()` and `.lower()` methods convert all characters of a string to uppercase and lowercase, respectively. The `.rstrip()`, `.lstrip()`, and `.strip()` methods remove whitespace from strings, and the `.startswith()` and `.endswith()` methods tell you if a string starts or ends with a given substring.

You also saw how to capture input from a user as a string using the `input()` function, and how to convert that input to a number using `int()` and `float()`. To convert numbers and other objects to strings, you use `str()`.

Finally, you saw how to use the `.find()` and `.replace()` methods to find the location of a substring and replace a substring with a new string.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-strings

Additional Resources

To learn more, check out the following resources:

- “Python String Formatting Best Practices”
- “Splitting, Concatenating, and Joining Strings in Python”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 5

Numbers and Math

You don't need to be a math whiz to program well. The truth is, few programmers need to know more than basic algebra.

Of course, how much math you need to know depends on the application you're working on. In general, the level of math required to be a programmer is lower than you might expect.

Although math and computer programming aren't as correlated as some people might believe, numbers are an integral part of any programming language, and Python is no exception.

In this chapter, you'll learn how to:

- Create integers and floating-point numbers
- Round numbers to a given number of decimal places
- Format and display numbers in strings

Let's get started!

5.1 Integers and Floating-Point Numbers

Python has three built-in numeric data types: integers, floating-point numbers, and complex numbers. In this section, you'll learn about integers and floating-point numbers, which are the two most commonly used number types. You'll learn about complex numbers in section 5.7.

Integers

An **integer** is a whole number with no decimal places. For example, 1 is an integer, but 1.0 isn't. The name for the integer data type is `int`, which you can see with `type()`:

```
>>> type(1)
<class 'int'>
```

You can create an integer by typing the desired number. For instance, the following assigns the integer 25 to the variable `num`:

```
>>> num = 25
```

When you create an integer like this, the value 25 is called an **integer literal** because the integer is literally typed into the code.

In chapter 4, you learned how to convert a string containing an integer to a number using `int()`. For example, the following converts the string "25" to the integer 25:

```
>>> int("25")
25
```

`int("25")` is not an integer literal because the integer value is created from a string.

When you write large numbers by hand, you typically group digits into groups of three separated by a comma or a decimal point. The number 1,000,000 is a lot easier to read than 1000000.

In Python, you can't use commas to group digits in integer literals, but you can use underscores (_). Both of the following are valid ways to represent the number one million as an integer literal:

```
>>> 1000000
```

```
1000000
```

```
>>> 1_000_000
```

```
1000000
```

There's no limit to how large an integer can be, which might be surprising considering that computers have a finite amount of memory. Try typing the largest number you can think of into IDLE's interactive window. Python can handle it with no problem!

Floating-Point Numbers

A **floating-point number**, or **float** for short, is a number with a decimal place. 1.0 is a floating-point number, as is -2.75. The name of the floating-point data type is **float**:

```
>>> type(1.0)
```

```
<class 'float'>
```

Like integers, floats can be created from **floating-point literals** or by converting a string to a float with **float()**:

```
>>> float("1.25")
```

```
1.25
```

There are three ways to represent a floating-point literal. Each of the following creates a floating-point literal with a value of one million:

```
>>> 1000000.0
```

```
1000000.0
```

```
>>> 1_000_000.0
```

```
1000000.0
```

```
>>> 1e6
```

```
1000000.0
```

The first two ways are similar to the two techniques for creating integer literals. The third approach uses **E notation** to create a float literal.

Note

E notation is short for **exponential notation**. You may have seen this notation used by calculators to represent numbers that are too big to fit on the screen.

To write a float literal in E notation, type a number followed by the letter e and then another number. Python takes the number to the left of the e and multiplies it by 10 raised to the power of the number after the e. So 1e6 is equivalent to 1×10^6 .

Python also uses E notation to display large floating-point numbers:

```
>>> 2000000000000000000.0
```

```
2e+17
```

The float 2000000000000000000.0 gets displayed as 2e+17. The + sign indicates that the exponent 17 is a positive number. You can also use negative numbers as the exponent:

```
>>> 1e-4
```

```
0.0001
```

The literal `1e-4` is interpreted as 10 raised to the power -4 , which is $1/10000$, or 0.0001 .

Unlike integers, floats do have a maximum size. The maximum floating-point number depends on your system, but something like `2e400` ought to be well beyond most machines' capabilities. `2e400` is 2×10^{400} , which is far more than the [total number of atoms in the universe!](#)

When you reach the maximum floating-point number, Python returns a special float value, `inf`:

```
>>> 2e400
inf
```

`inf` stands for infinity, and it just means that the number you've tried to create is beyond the maximum floating-point value allowed on your computer. The type of `inf` is still `float`:

```
>>> n = 2e400
>>> n
inf
>>> type(n)
<class 'float'>
```

Python also uses `-inf`, which stands for negative infinity and represents a negative floating-point number that is beyond the minimum floating-point number allowed on your computer:

```
>>> -2e400
-inf
```

You probably won't come across `inf` and `-inf` very often as a programmer unless you regularly work with extremely large numbers.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that creates two variables, `num1` and `num2`. Both `num1` and `num2` should be assigned the integer literal `25000000`, one written with underscores and one without. Print `num1` and `num2` on two separate lines.
2. Write a program that assigns the floating-point literal `175000.0` to the variable `num` using E notation and then prints `num` in the interactive window.
3. In IDLE’s interactive window, try to find the smallest exponent `N` for which `2e<N>`, where `<N>` is replaced with your number, returns `inf`.

5.2 Arithmetic Operators and Expressions

In this section, you’ll learn how to do basic arithmetic, such as addition, subtraction, multiplication, and division, with numbers in Python. Along the way, you’ll learn some conventions for writing mathematical expressions in code.

Addition

Addition is performed with the `+` operator:

```
>>> 1 + 2  
3
```

The two numbers on either side of the `+` operator are called **operands**. In the above example, both operands are integers, but operands don’t need to be the same type.

You can add an `int` to a `float` with no problem:

```
>>> 1.0 + 2  
3.0
```

Notice that the result of `1.0 + 2` is `3.0`, which is a `float`. Anytime a `float` is added to a number, the result is another `float`. Adding two integers together always results in an `int`.

Note

[PEP 8 recommends](#) separating both operands from an operator with a space.

Python can evaluate `1+1` just fine, but `1 + 1` is the preferred format because it's generally easier to read. This rule of thumb applies to all the operators in this section.

Subtraction

To subtract two numbers, just put a `-` operator between them:

```
>>> 1 - 1  
0  
  
>>> 5.0 - 3  
2.0
```

Just like adding two integers, subtracting two integers always results in an `int`. Whenever one of the operands is a `float`, the result is also a `float`.

The `-` operator is also used to denote negative numbers:

```
>>> -3  
-3
```

You can subtract a negative number from another number, but as you can see below, this can sometimes look confusing:

```
>>> 1 - -3
```

```
4
```

```
>>> 1 --3
```

```
4
```

```
>>> 1- -3
```

```
4
```

```
>>> 1--3
```

```
4
```

Of the four examples above, the first is the most PEP 8 compliant. That said, you can surround `-3` with parentheses to make it even clearer that the second `-` is modifying `3`:

```
>>> 1 - (-3)
```

```
4
```

Using parentheses is a good idea because it makes your code more explicit. Computers execute code, but humans read code. Anything you can do to make your code easier to read and understand is a good thing.

Multiplication

To multiply two numbers, use the `*` operator:

```
>>> 3 * 3
```

```
9
```

```
>>> 2 * 8.0
```

```
16.0
```

The type of number you get from multiplication follows the same rules as addition and subtraction. Multiplying two integers results in an `int`, and multiplying a number with a `float` results in a `float`.

Division

The `/` operator is used to divide two numbers:

```
>>> 9 / 3  
3.0  
  
>>> 5.0 / 2  
2.5
```

Unlike addition, subtraction, and multiplication, division with the `/` operator always returns a `float`. If you want to make sure that you get an integer after dividing two numbers, **you can use `int()` to convert the result:**

```
>>> int(9 / 3)  
3
```

Keep in mind that `int()` discards any fractional part of the number:

```
>>> int(5.0 / 2)  
2
```

`5.0 / 2` returns the floating-point number `2.5`, and `int(2.5)` returns the integer `2` with the `.5` removed.

Integer Division

If writing `int(5.0 / 2)` seems a little long-winded to you, Python provides a second division operator called the **integer division operator** (`//`), also known as the **floor division** operator:

```
>>> 9 // 3  
3  
  
>>> 5.0 // 2  
2.0  
  
>>> -3 // 2  
-2
```

The `//` operator first divides the number on the left by the number on the right and then rounds down to an integer. This might not give the value you expect when one of the numbers is negative.

For example, `-3 // 2` returns `-2`. First, `-3` is divided by `2` to get `-1.5`. Then `-1.5` is rounded down to `-2`. On the other hand, `3 // 2` returns `1` because both numbers are positive.

The above example also illustrates that `//` returns a floating-point number when one of the operands is a `float`. This is why `9 // 3` returns the integer `3`, and `5.0 // 2` returns the `float` `2.0`.

Let's see what happens when you try to divide a number by `0`:

```
>>> 1 / 0  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: division by zero
```

Python gives you a `ZeroDivisionError`, letting you know that you just tried to break a fundamental rule of the universe.

Exponents

You can raise a number to a power using the `**` operator:

```
>>> 2 ** 2
```

```
4
```

```
>>> 2 ** 3
```

```
8
```

```
>>> 2 ** 4
```

```
16
```

Exponents don't have to be integers. They can also be floats:

```
>>> 3 ** 1.5
```

```
5.196152422706632
```

```
>>> 9 ** 0.5
```

```
3.0
```

Raising a number to the power of 0.5 is the same as taking the square root, but notice that even though the square root of 9 is an integer, Python returns the float 3.0.

For positive operands, the `**` operator returns an `int` if both operands are integers and a `float` if any one of the operands is a floating-point number.

You can also raise numbers to negative powers:

```
>>> 2 ** -1
```

```
0.5
```

```
>>> 2 ** -2
```

```
0.25
```

Raising a number to a negative power is the same as dividing 1 by the number raised to the positive power. So, 2^{-1} is the same as $1 / (2^1)$, which is the same as $1 / 2$, or 0.5. Similarly, 2^{-2} is the same as $1 / (2^2)$, which is the same as $1 / 4$, or 0.25.

The Modulus Operator

The % operator, or the **modulus**, returns the remainder of dividing the left operand by the right operand:

```
>>> 5 % 3  
2  
  
>>> 20 % 7  
6  
  
>>> 16 % 8  
0
```

3 divides 5 once with a remainder of 2, so $5 \% 3$ is 2. Similarly, 7 divides 20 twice with a remainder of 6. In the last example, 16 is divisible by 8, so $16 \% 8$ is 0. Anytime the number to the left of % is divisible by the number to the right, the result is 0.

One of the most common uses of % is to determine whether one number is divisible by another. For example, a number n is even if and only if $n \% 2$ is 0. What do you think $1 \% 0$ returns? Let's try it out:

```
>>> 1 % 0  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
ZeroDivisionError: integer division or modulo by zero
```

This makes sense because $1 \% 0$ gives the remainder of dividing 1 by 0. But you can't divide 1 by 0, so Python raises a ZeroDivisionError.

Note

When you work in IDLE’s interactive window, errors like `zeroDivisionError` don’t cause much of a problem. The error is displayed and a new prompt pops up, allowing you to continue writing code.

However, when Python encounters an error while running a program, the execution stops. In other words, the program **crashes**. In chapter 8, you’ll learn how to handle errors so that your programs don’t crash unexpectedly.

Things get a little tricky when you use the `%` operator with negative numbers:

```
>>> 5 % -3  
-1  
  
>>> -5 % 3  
1  
  
>>> -5 % -3  
-2
```

Although potentially shocking at first glance, these results are the product of a well-defined behavior in Python. To calculate the remainder r of dividing a number x by a number y , Python uses the equation $r = x - (y * (x // y))$.

For example, to find $5 \% -3$, Python first finds $(5 // -3)$. Since $5 / -3$ is about -1.67 , that means $5 // -3$ is -2 . Now Python multiplies that by -3 to get 6 . Finally, Python subtracts 6 from 5 to get -1 .

Arithmetic Expressions

You can combine operators to form complex expressions. An **expression** is a combination of numbers, operators, and parentheses that Python can compute, or **evaluate**, to return a value.

Here are some examples of arithmetic expressions:

```
>>> 2*3 - 1  
5  
  
>>> 4/2 + 2**3  
10.0  
  
>>> -1 + (-3*2 + 4)  
-3
```

The rules for evaluating expressions are the same as in everyday arithmetic. In school, you probably learned these rules under the name *order of operations*.

The *, /, //, and % operators all have equal **precedence**, or priority, in an expression, and each of these has a higher precedence than the + and - operators. This is why $2*3 - 1$ returns 5 and not 4. $2*3$ is evaluated first, because * has higher precedence than the - operator.

You may notice that the expressions in the previous example do not follow the rule for putting a space on either side of all of the operators. PEP 8 says the following about whitespace in complex expressions:

If operators with different priorities are used, consider adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator.

— PEP 8, “Other Recommendations”

Another good practice is to use parentheses to indicate the order in which operations should be performed, even if the parentheses aren’t necessary. For instance, $(2 * 3) - 1$ is potentially clearer than $2*3 - 1$.

5.3 Challenge: Perform Calculations on User Input

Write a program called `exponent.py` that receives two numbers from the user and displays the first number raised to the power of the second number.

Here's sample run of what the program should look like, including example input from the user:

```
Enter a base: 1.2
Enter an exponent: 3
1.2 to the power of 3 = 1.7279999999999998
```

Keep the following in mind:

1. Before you can do anything with the user's input, you'll have to assign both calls to `input()` to new variables.
2. `input()` returns a string, so you'll need to convert the user's input into numbers to do arithmetic.
3. You can use an f-string to print the result.
4. You can assume that the user will enter actual numbers as input.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

5.4 Make Python Lie to You

What do you think `0.1 + 0.2` is? The answer is `0.3`, right? Let's see what Python has to say about it. Try this out in the interactive window:

```
>>> 0.1 + 0.2
0.3000000000000004
```

Well, that's . . . *almost* right. What in the heck is going on here? Is this a bug in Python?

No, it's not a bug! It's a **floating-point representation error**, and it has nothing to do with Python. It's related to the way floating-point numbers are stored in a computer's memory.

The number 0.1 can be represented as the fraction $1/10$. Both the number 0.1 and its fraction $1/10$ are **decimal representations**, or **base-10 representations**. Computers, however, store floating-point numbers in base-2 representation, more commonly called **binary representation**.

When represented in binary, something familiar yet possibly unexpected happens to the decimal number 0.1. The fraction $1/3$ has no finite decimal representation. That is, $1/3 = 0.333\dots$ with infinitely many 3s after the decimal point. The same thing happens to the fraction $1/10$ in binary.

The binary representation of $1/10$ is the following infinitely repeating fraction:

```
0.00011001100110011001100110011...
```

Computers have finite memory, so the number 0.1 must be stored as an approximation and not as its true value. The approximation that gets stored is slightly higher than the actual value and looks like this:

```
0.100000000000000055511151231257827021181583404541015625
```

You may have noticed, however, that when asked to print 0.1, Python prints 0.1 and not the approximated value above:

```
>>> 0.1  
0.1
```

Python doesn't just chop off the digits in the binary representation for 0.1. What actually happens is a little more subtle.

Because the approximation of 0.1 in binary is just that—an approximation—it is entirely possible that more than one decimal number has the same binary approximation.

For example, both `0.1` and `0.1000000000000001` have the same binary approximation. Python prints out the shortest decimal number that shares the approximation.

This explains why, in the first example of this section, `0.1 + 0.2` doesn't equal `0.3`. Python adds together the binary approximations for `0.1` and `0.2`, which gives a number that is *not* the binary approximation for `0.3`.

If all this is starting to make your head spin, don't worry! Unless you're writing programs for finance or scientific computing, you don't need to worry about the imprecision of floating-point arithmetic.

5.5 Math Functions and Number Methods

Python has a few built-in functions that you can use to work with numbers. In this section, you'll learn about three of the most common:

1. `round()`, for rounding numbers to some number of decimal places
2. `abs()`, for getting the absolute value of a number
3. `pow()`, for raising a number to some power

You'll also learn about a method you can use with floating-point numbers to check whether they have an integer value.

Let's go!

The `round()` function

You can use `round()` to round a number to the nearest integer:

```
>>> round(2.3)
```

```
2
```

```
>>> round(2.7)
```

```
3
```

`round()` has some unexpected behavior when the number ends in .5:

```
>>> round(2.5)
```

```
2
```

```
>>> round(3.5)
```

```
4
```

2.5 is rounded down to 2, and 3.5 is rounded up to 4. Most people expect a number that ends in .5 to be rounded up, so let's take a closer look at what's going on here.

Python 3 rounds numbers according to a strategy called **rounding ties to even**. A **tie** is any number whose last digit is five. 2.5 and 3.1415 are ties, but 1.37 is not.

When you round ties to even, you first look at the digit one decimal place to the left of the last digit in the tie. If that digit is even, then you round down. If the digit is odd, then you round up. That's why 2.5 rounds down to 2 and 3.5 rounds up to 4.

Note

Rounding ties to even is the rounding strategy recommended for floating-point numbers by the [IEEE](#) (Institute of Electrical and Electronics Engineers) because it helps limit the impact that rounding has on operations involving lots of numbers.

The IEEE maintains a standard called [IEEE 754](#) for dealing with floating-point numbers on a computer. It was published in 1985 and is still commonly used by hardware manufacturers.

You can round a number to a given number of decimal places by passing a second argument to `round()`:

```
>>> round(3.14159, 3)
```

```
3.142
```

```
>>> round(2.71828, 2)
```

```
2.72
```

The number 3.14159 is rounded to three decimal places to get 3.142, and the number 2.71828 is rounded to two decimal places to get 2.72.

The second argument of `round()` must be an integer. If it isn't, then Python raises a `TypeError`:

```
>>> round(2.65, 1.4)
```

```
Traceback (most recent call last):
```

```
  File "<pyshell#0>", line 1, in <module>
    round(2.65, 1.4)
```

```
TypeError: 'float' object cannot be interpreted as an integer
```

Sometimes `round()` doesn't get the answer quite right:

```
>>> # Expected value: 2.68
```

```
>>> round(2.675, 2)
```

```
2.67
```

2.675 is a tie because it lies exactly halfway between the numbers 2.67 and 2.68. Since Python rounds ties to the nearest even number, you would expect `round(2.675, 2)` to return 2.68, but it returns 2.67 instead. This error is the result of a floating-point representation error, not a bug in `round()`.

Dealing with floating-point numbers can be frustrating, but this frustration isn't specific to Python. All languages that implement the IEEE floating-point standard have the same issues, including C/C++, Java, and JavaScript.

In most cases, though, the little errors encountered with floating-point numbers are negligible, and the results of `round()` are perfectly useful.

The `abs()` Function

The **absolute value** of a number n is just n if n is positive and $-n$ if n is negative. For example, the absolute value of 3 is 3, while the absolute value of -5 is 5.

To get the absolute value of a number in Python, you use `abs()`:

```
>>> abs(3)
3

>>> abs(-5.0)
5.0
```

`abs()` always returns a positive number of the same type as its argument. That is, the absolute value of an integer is always a positive integer, and the absolute value of a float is always a positive float.

The `pow()` Function

In section 5.2, you learned how to raise a number to a power using the `**` operator. You can also use `pow()` to achieve the same result.

`pow()` takes two arguments. The first argument is the **base**, or the number to be raised to a power, and the second argument is the **exponent**, or the power to which the number is to be raised.

For example, the following uses `pow()` to raise 2 to the exponent 3:

```
>>> pow(2, 3)
8
```

Just like with `**`, the exponent in `pow()` can be negative:

```
>>> pow(2, -2)
0.25
```

So, what's the difference between `**` and `pow()`?

The `pow()` function accepts an optional third argument that computes the first number raised to the power of the second number, then takes the modulo with respect to the third number. In other words, `pow(x, y, z)` is equivalent to `(x ** y) % z`.

Here's an example in which `x = 2`, `y = 3`, and `z = 2`:

```
>>> pow(2, 3, 2)
0
```

First, 2 is raised to the power 3 to get 8. Then $8 \% 2$ is calculated, which is 0 because 2 divides 8 with no remainder.

Check If a Float Is Integral

In chapter 3 you learned about string methods like `.lower()`, `.upper()`, and `.find()`. Integers and floating-point numbers also have methods.

Number methods aren't used very often, but there is one that can be useful. Floating-point numbers have an `.is_integer()` method that returns `True` if the number is **integral**—meaning it has no fractional part—and otherwise returns `False`:

```
>>> num = 2.5
>>> num.is_integer()
False

>>> num = 2.0
>>> num.is_integer()
True
```

The `.is_integer()` method can be useful for validating user input. For example, if you were writing an online ordering app for a pizzeria, then you would want to check that the quantity of pizzas the customer inputs is a whole number. You'll learn how to do these kinds of checks in chapter 8.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that asks the user to input a number and then displays that number rounded to two decimal places. When run, your program should look like this:

```
Enter a number: 5.432
5.432 rounded to 2 decimal places is 5.43
```

2. Write a program that asks the user to input a number and then displays the absolute value of that number. When run, your program should look like this:

```
Enter a number: -10
The absolute value of -10 is 10.0
```

3. Write a program that asks the user to input two numbers by using `input()` twice, then displays whether the difference between those two numbers is an integer. When run, your program should look like this:

```
Enter a number: 1.5
Enter another number: .5
The difference between 1.5 and .5 is an integer? True!
```

If the user inputs two numbers whose difference is not integral, then the output should look like this:

```
Enter a number: 1.5
Enter another number: 1.0
The difference between 1.5 and 1.0 is an integer? False!
```

5.6 Print Numbers in Style

Displaying numbers to a user requires inserting numbers into a string. In chapter 3, you learned how to do this with f-strings by surrounding a variable assigned to a number with curly braces:

```
>>> n = 7.125
>>> f"The value of n is {n}"
'The value of n is 7.125'
```

Those curly braces support a simple [formatting language](#) that you can use to alter the appearance of the value in the final formatted string.

For example, to format the value of `n` in the above example to two decimal places, replace the contents of the curly braces in the f-string with `{n:.2f}`:

```
>>> n = 7.125
>>> f"The value of n is {n:.2f}"
'The value of n is 7.12'
```

The colon (`:`) after the variable `n` indicates that everything after it is part of the formatting specification. In this example, the formatting specification is `.2f`.

The `.2` in `.2f` rounds the number to two decimal places, and the `f` tells Python to display `n` as a **fixed-point number**. This means that the number is displayed with exactly two decimal places, even if the original number has fewer decimal places.

When `n = 7.125`, the result of `{n:.2f}` is `7.12`. Just like with `round()`, Python rounds ties to even when formatting numbers inside strings. So, if you replace `n = 7.125` with `n = 7.126`, then the result of `{n:.2f}` is `7.13`:

```
>>> n = 7.126
>>> f"The value of n is {n:.2f}"
'The value of n is 7.13'
```

To round to one decimal place, replace .2 with .1:

```
>>> n = 7.126
>>> f"The value of n is {n:.1f}"
'The value of n is 7.1'
```

When you format a number as fixed point, it's always displayed with the precise number of decimal places that you specify:

```
>>> n = 1
>>> f"The value of n is {n:.2f}"
'The value of n is 1.00'
>>> f"The value of n is {n:.3f}"
'The value of n is 1.000'
```

You can insert commas to group the integer part of large numbers by the thousands with the , option:

```
>>> n = 1234567890
>>> f"The value of n is {n:,}"
'The value of n is 1,234,567,890'
```

To round to some number of decimal places and also group by thousands, put the , before the . in your formatting specification:

```
>>> n = 1234.56
>>> f"The value of n is {n:,.2f}"
'The value of n is 1,234.56'
```

The specifier ,.2f is useful for displaying currency values:

```
>>> balance = 2000.0
>>> spent = 256.35
>>> remaining = balance - spent

>>> f"After spending ${spent:.2f}, I was left with ${remaining:,.2f}"
'After spending $256.35, I was left with $1,743.65'
```

Another useful option is %, which is used to display percentages. The % option multiplies a number by 100 and displays it in fixed-point format, followed by a percentage sign.

The % option should always go at the end of your formatting specification, and you can't mix it with the f option. For example, .1% displays a number as a percentage with exactly one decimal place:

```
>>> ratio = 0.9
>>> f"Over {ratio:.1%} of Pythonistas say 'Real Python rocks!''"
"Over 90.0% of Pythonistas say 'Real Python rocks!'"

>>> # Display percentage with 2 decimal places
>>> f"Over {ratio:.2%} of Pythonistas say 'Real Python rocks!''"
"Over 90.00% of Pythonistas say 'Real Python rocks!'"
```

The formatting mini language is powerful and extensive. You've only seen the basics here. For more information, you are encouraged to read the [official documentation](#).

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Print the result of the calculation `3 ** .125` as a fixed-point number with three decimal places.
2. Print the number `150000` as currency with the thousands grouped with commas. Currency should be displayed with two decimal places.
3. Print the result of `2 / 10` as a percentage with no decimal places. The output should look like `20%`.

5.7 Complex Numbers

Python is one of the few programming languages that provides built-in support for complex numbers. While complex numbers don't often come up outside the domains of scientific computing and computer graphics, Python's support for them is one of its strengths.

Note

Feel free to skip this section if you have no interest in working with complex numbers in Python. No other part of the book depends on the information in this section.

If you've ever taken a precalculus or higher-level algebra math class, then you may remember that a complex number is a number with two distinct components: a **real** part and an **imaginary** part.

To create a complex number in Python, you simply write the real part, then a plus sign, then the imaginary part with the letter *j* at the end:

```
>>> n = 1 + 2j
```

When you inspect the value of *n*, you'll notice that Python wraps the number with parentheses:

```
>>> n  
(1+2j)
```

This convention helps eliminate any confusion that the displayed output may represent a string or a mathematical expression.

Imaginary numbers come with two properties, `.real` and `.imag`, that return the real and imaginary components of the number, respectively:

```
>>> n.real
```

```
1.0
```

```
>>> n.imag
```

```
2.0
```

Notice that Python returns both the real and imaginary components as floats, even though they were specified as integers.

Complex numbers also have a `.conjugate()` method that returns the complex conjugate of the number:

```
>>> n.conjugate()
```

```
(1-2j)
```

For any complex number, its **conjugate** is the complex number with the same real part and an imaginary part that is the same in absolute value but with the opposite sign. So in this case, the complex conjugate of $1 + 2j$ is $1 - 2j$.

Note

The `.real` and `.imag` properties don't need parentheses after them like `.conjugate()` does.

The `.conjugate()` method is a function that performs an action on a complex number, whereas `.real` and `.imag` don't perform any action—they just return some information about the number.

The distinction between methods and properties is an important aspect of **object-oriented programming**, which you'll learn about in chapter 10.

Except for the floor division operator (`//`), all the arithmetic operators that work with floats and integers will also work with complex numbers.

Since this isn't a math book, we won't discuss the mechanics of complex arithmetic. Instead, here are some examples of using complex numbers with arithmetic operators:

```
>>> a = 1 + 2j
>>> b = 3 - 4j

>>> a + b
(4-2j)

>>> a - b
(-2+6j)

>>> a * b
(11+2j)

>>> a ** b
(932.1391946432212+95.9465336603415j)

>>> a / b
(-0.2+0.4j)

>>> a // b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't take floor of complex number.
```

Interestingly, although not surprising from a mathematical point of view, `int` and `float` objects also have `.real` and `.imag` properties as well as the `.conjugate()` method:

```
>>> x = 42
>>> x.real
42
>>> x.imag
0
```

```
>>> x.conjugate()
42

>>> y = 3.14
>>> y.real
3.14
>>> y.imag
0.0
>>> y.conjugate()
3.14
```

For floats and integers, `.real` and `.conjugate()` always return the number itself, and `.imag` always returns 0. One thing to notice, however, is that `n.real` and `n.imag` return an integer if `n` is an integer and a float if `n` is a float.

Now that you've seen the basics of complex numbers, you might be wondering when you would ever need to use them. If you're learning Python for web development, data science, or general-purpose programming, the truth is that you may *never* need to use complex numbers.

On the other hand, complex numbers are important in domains such as scientific computing and computer graphics. If you ever work in those domains, then you may find Python's built-in support for complex numbers useful.

5.8 Summary and Additional Resources

In this chapter, you learned all about working with numbers in Python. You saw that there are two basic types of numbers—integers and floating-point numbers—and that Python also has built-in support for complex numbers.

First, you saw how to do basic arithmetic with numbers using the +, -, *, /, and % operators. You learned how to write arithmetic expressions and the best practices in [PEP 8](#) for formatting arithmetic expressions in your code.

Then you learned about floating-point numbers and how they may not always be 100 percent accurate. This limitation has nothing to do with Python. It's a fact of modern-day computing related to the way floating-point numbers are stored in a computer's memory.

Next, you saw how to round numbers to a given decimal place with `round()` and learned that `round()` rounds ties to even, which is different from the way most people learned to round numbers in school. You also saw numerous ways to format numbers for display.

Finally, you learned about Python's built-in support for complex numbers.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-numbers

Additional Resources

To learn more, check out these resources:

- “[Basic Data Types in Python](#)”
- “[How to Round Numbers in Python](#)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 6

Functions and Loops

Functions are the building blocks of almost every Python program. They're where the real action takes place!

You've already seen how to use several functions, including `print()`, `len()`, and `round()`. These are all **built-in functions** because they come built into the Python language itself. You can also create **user-defined functions** that perform specific tasks.

Functions break code into smaller chunks and are great for defining actions that a program will execute several times throughout your code. Instead of writing the same code each time the program needs to perform the same task, just call the function!

But sometimes you do need to repeat some code several times in a row. This is where **loops** come in.

In this chapter, you'll learn:

- How to create user-defined functions
- How to write `for` and `while` loops
- What scope is and why it's important

Let's dive in!

6.1 What Is a Function, Really?

In the past few chapters, you used the functions `print()` and `len()` to display text and determine the length of a string. But what is a function, really?

In this section, you'll take a closer look at `len()` to learn more about what a function is and how it's executed.

Functions Are Values

One of the most important properties of functions in Python is that functions are values that can be assigned to a variable.

In IDLE's interactive window, inspect the name `len` by typing the following at the prompt:

```
>>> len  
<built-in function len>
```

Python tells you that `len` is a built-in function. Just like integer values have a type called `int`, and strings have a type called `str`, function values also have a type:

```
>>> type(len)  
<class 'builtin_function_or_method'>
```

The name `len`, though, can be assigned to a different value if you like:

```
>>> len = "I'm not the len you're looking for."  
>>> len  
"I'm not the len you're looking for."
```

Now `len` has a string value, and you can verify that its type is `str` with `type()`:

```
>>> type(len)  
<class 'str'>
```

Even though you can change the value of the name `len`, it's usually a bad idea to do so. Changing the value of `len` can make your code confusing because it's easy to mistake the new `len` for the built-in function. The same goes for any built-in function.

Important

If you typed in the previous code examples, then **you no longer have access to the built-in `len` function in IDLE**.

You can get it back with the following code:

```
>>> del len
```

The `del` keyword is used to unassign a variable from a value. `del` stands for delete, but it doesn't delete the value. Instead, it detaches the name from the value and deletes the name.

Normally, after using `del`, trying to use the deleted variable name raises a `NameError`. But in this case, the name `len` doesn't get deleted:

```
>>> len
<built-in function len>
```

Because `len` is a built-in function name, it gets reassigned to the original function value.

So what does all of this show you? Functions have names, but those names are only loosely tied to the function and can be assigned to different values.

When you write your own functions, be careful not to give them names used by built-in functions.

How Python Executes Functions

Now let's take a closer look at how Python executes a function.

The first thing to notice is that you can't execute a function by typing just its name. You must **call** the function to tell Python to actually execute it.

Let's look at how this works with `len()`:

```
>>> # Typing just the name doesn't execute the function.  
>>> # IDLE inspects the variable as usual.  
>>> len  
<built-in function len>  
  
>>> # Use parentheses to call the function.  
>>> len()  
Traceback (most recent call last):  
  File "<pyshell#3>", line 1, in <module>  
    len()  
TypeError: len() takes exactly one argument (0 given)
```

In this example, Python raises a `TypeError` when `len()` is called because `len()` expects an argument.

An **argument** is a value that gets **passed** to the function as input. Some functions can be called with no arguments, and some can take as many arguments as you like. `len()` requires exactly one argument.

When a function is done executing, it **returns** a value as output. The return value usually—but not always—depends on the values of any arguments passed to the function.

The process for executing a function can be summarized in three steps:

1. The function is **called**, and any arguments are passed to the function as input.
2. The function **executes**, and some action is performed with the arguments.
3. The function **returns**, and the original function call is replaced with the return value.

Let's look at this in practice and see how Python executes the following line of code:

```
>>> num_letters = len("four")
```

First, `len()` is called with the argument "four". The length of the string "four" is calculated, which is the number 4. Then `len()` returns the number 4 and replaces the function call with the value.

You can imagine that, after the function executes, the line of code looks like this:

```
>>> num_letters = 4
```

Then Python assigns the value 4 to `num_letters` and continues executing any remaining lines of code in the program.

Functions Can Have Side Effects

You've learned how to call functions and that they return a value when they're done executing. Sometimes, though, functions do more than just return a value.

When a function changes or affects something external to itself, it is said to have a **side effect**. You've already seen one function with a side effect: `print()`.

When you call `print()` with a string argument, the string is displayed in the Python shell as text. But `print()` doesn't return any text as a value.

To see what `print()` returns, you can assign the return value of `print()` to a variable:

```
>>> return_value = print("What do I return?")
What do I return?
>>> return_value
>>>
```

When you assign `print("What do I return?")` to `return_value`, the string "What do I return?" is displayed. However, when you inspect the value of `return_value`, nothing is shown.

`print()` returns a special value called `None` that indicates the absence of data. `None` has a type called `NoneType`:

```
>>> type(return_value)
<class 'NoneType'>
>>> print(return_value)
None
```

When you call `print()`, the text that gets displayed is not the return value. It's a side effect of `print()`.

6.2 Write Your Own Functions

As you write more complex programs, you may find that you need to repeatedly use the same few lines of code. For instance, you might need to calculate the same formula with different values several times in your code.

You might be tempted to copy and paste similar code to other parts of your program and modify it as needed, but this is usually a bad idea! If you find a mistake in code that's been copied and pasted all over the place, then you'll have to apply the fix everywhere the code was copied. That's a lot of work!

In this section, you'll learn how to define your own functions so that you can avoid repeating yourself when you need to reuse code.

The Anatomy of a Function

Every function has two parts:

1. The **function signature** defines the name of the function and any inputs it expects.

2. The **function body** contains the code that runs every time the function is used.

Let's start by writing a function that takes two numbers as input and returns their product. Here's what this function might look like, with the signature and body identified in comments:

```
def multiply(x, y): # Function signature
    # Function body
    product = x * y
    return product
```

It might seem odd to make a function for something as simple as the `*` operator. In fact, `multiply()` probably isn't a function that you would write in a real-world scenario. But it makes a great first example for understanding how functions are created!

Important

When you define a function in IDLE's interactive window, you need to press `Enter` twice after the line containing `return` for Python to register the function:

```
>>> def multiply(x, y):
...     product = x * y
...     return product
... # <--- Hit Enter a second time here.
>>>
```

Let's break the function down to see what's going on.

The Function Signature

The first line of code in a function is called the **function signature**. It always starts with the `def` keyword, which is short for `define`.

Let's look more closely at the signature of `multiply()`:

```
def multiply(x, y):
```

The function signature has four parts:

1. The `def` keyword
2. The function name, `multiply`
3. The parameter list, `(x, y)`
4. A colon (`:`) at the end of the line

When Python reads a line beginning with the `def` keyword, it creates a new function. The function is assigned to a variable with the same name as the function name.

Note

Since function names become variables, they must follow the same rules for variable names that you learned in chapter 3.

So, a function name can only contain numbers, letters, and underscores, and it must not begin with a number.

The parameter list is a list of parameter names surrounded by parentheses. It defines the function's expected inputs. `(x, y)` is the parameter list for `multiply()`. It creates two parameters, `x` and `y`.

A **parameter** is sort of like a variable, except that it has no value. It's a placeholder for actual values that will be provided whenever the function is called with one or more arguments.

Code in the function body can use parameters as if they were variables with real values. For example, the function body may contain a line of code with the expression `x * y`.

Since `x` and `y` have no value, `x * y` has no value. Python saves the expression as a template and fills in the missing values when the function is executed.

A function can have any number of parameters, including no parameters at all!

The Function Body

The **function body** is the code that runs whenever the function is used in your program. Here's the function body for `multiply()`:

```
def multiply(x, y):
    # Function body
    product = x * y
    return product
```

`multiply` is a pretty simple function. Its body has only two lines of code!

The first line of the function body creates a variable called `product` and assigns to it the value `x * y`. Since `x` and `y` have no values yet, this line is really a template for the value that will be assigned to `product` when the function is executed.

The second line in the function body is called a **return statement**. It starts with the `return` keyword and is followed by the variable `product`. When Python reaches the return statement, it stops running the function and returns the value of `product`.

Notice that both lines of code in the function body are indented. This is vitally important! Every line that is indented below the function signature is understood to be part of the function body.

For instance, the `print()` function in the following example is not part of the function body because it's not indented:

```
def multiply(x, y):
    product = x * y
    return product

print("Where am I?")  # Not in the function body
```

If you indent `print()`, then it becomes part of the function body even if there's a blank line between `print()` and the previous line:

```
def multiply(x, y):
    product = x * y
    return product

    print("Where am I?") # In the function body
```

There is one rule that you must follow when indenting code in a function's body: every line must be indented by the same number of spaces.

Try saving the following code to a file called `multiply.py` and running it from IDLE:

```
def multiply(x, y):
    product = x * y
    return product # Indented with one extra space
```

IDLE won't run the code! A dialog box appears with the error "unexpected indent." Python expects the `return` statement to be indented the same number of spaces as the line above it.

Another error occurs when a line of code is indented less than the line above it, and the indentation doesn't match any previous lines. Modify the `multiply.py` file to look like this:

```
def multiply(x, y):
    product = x * y
    return product # Indented less than previous line
```

Now save and run the file. IDLE stops it with the error `unindent does not match any outer indentation level`. The `return` statement isn't indented the same number of spaces as any other line in the function body.

Note

Although Python has no rules for how many spaces to indent code in a function body, PEP 8 recommends indenting with [four spaces](#).

We follow this convention throughout this book.

Once Python executes a `return` statement, the function stops running and returns the value. If any code appears below the `return` statement and is indented like it's part of the function body, then it will never run.

For instance, `print()` will never be executed in the following function:

```
def multiply(x, y):
    product = x * y
    return product
    print("You can't see me!")
```

This version of `multiply()` never prints the string "You can't see me!"

Calling a User-Defined Function

You call a user-defined function just like any other function: type the function name followed by a list of arguments enclosed by parentheses.

For instance, to call `multiply()` with the argument 2 and 4, just type the following:

```
multiply(2, 4)
```

Unlike built-in functions, user-defined functions are not available until they've been defined with the `def` keyword. You must define the function before you call it.

Type the following program into IDLE's editor window:

```
num = multiply(2, 4)
print(num)

def multiply(x, y):
    product = x * y
    return product
```

Save the file and press **F5**. Since `multiply()` is called before it's defined, Python doesn't recognize the name `multiply` and raises a `NameError`:

```
Traceback (most recent call last):
  File "C:Usersdaveamultiply.py", line 1, in <module>
    num = multiply(2, 4)
NameError: name 'multiply' is not defined
```

To fix the error, move the function definition to the top of the file:

```
def multiply(x, y):
    product = x * y
    return product

num = multiply(2, 4)
print(num)
```

Save the file again and press **F5** to run the program. This time, the value 8 is displayed in the interactive window.

Functions With No Return Statement

All functions in Python return a value, even if that value is `None`. However, not all functions need a `return` statement.

For example, the following function is perfectly valid:

```
def greet(name):
    print(f"Hello, {name}!")
```

`greet()` has no return statement but works just fine:

```
>>> greet("Dave")
Hello, Dave!
```

Even though `greet()` has no return statement, it still returns a value:

```
>>> return_value = greet("Dave")
Hello, Dave!
>>> print(return_value)
None
```

The string "Hello, Dave!" is printed even when the result of `greet("Dave")` is assigned to a variable. If you weren't expecting to see "Hello, Dave!" printed, then you just experienced one of the issues with side effects. They can be unexpected!

When you create your own functions, you should always document what they do. That way, other developers can read the documentation to know how to use the functions and what to expect from them.

Documenting Your Functions

To get help with a function in IDLE's interactive window, you can use `help()`:

```
>>> help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

When you pass a variable name or function name to `help()`, it displays some useful information about the variable or function in question. In

this case, `help()` tells you that `len()` is a built-in function that returns the number of items in a container.

Note

A **container** is a special name for an object that contains other objects. A string is a container because it contains characters.

You'll learn about other container types in chapter 9.

Let's see what happens when you call `help()` on `multiply()`:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
```

`help()` displays the function signature, but there isn't any information about what the function does. To better document `multiply()`, we need to provide a docstring. A **docstring** is a triple-quoted string literal placed at the top of the function body.

Docstrings are used to document what a function does and what kinds of parameters it expects:

```
def multiply(x, y):
    """Return the product of two numbers x and y."""
    product = x * y
    return product
```

Rewrite your `multiply()` function with the docstring. Now you can use `help()` in the interactive window to see the docstring:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
    Return the product of two numbers x and y.
```

PEP 8 doesn't say much about docstrings, except that [every function should have one](#).

There are a number of standardized docstring formats, but we won't get into them here. Some general guidelines for writing docstrings can be found in [PEP 257](#).

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a function called `cube()` that takes one number parameter and returns the value of that number raised to the third power. Test the function by calling your `cube()` function on a few different numbers and displaying the results.
2. Write a function called `greet()` that takes one string parameter called `name` and displays the text "Hello <name>!", where `<name>` is replaced by the value of the `name` parameter.

6.3 Challenge: Convert Temperatures

Write a program called `temperature.py` that defines two functions:

1. `convert_cel_to_far()`, which takes one `float` parameter representing degrees Celsius and returns a `float` representing the same temperature in degrees Fahrenheit using the following formula:

$$F = C * 9/5 + 32$$

2. `convert_far_to_cel()`, which takes one `float` parameter representing degrees Fahrenheit and returns a `float` representing the same temperature in degrees Celsius using the following formula:

$$C = (F - 32) * 5/9$$

The program should do the following:

1. Prompt the user to enter a temperature in degrees Fahrenheit and then display the temperature converted to Celsius
2. Prompt the user to enter a temperature in degrees Celsius and then display the temperature converted to Fahrenheit
3. Display all converted temperatures rounded to two decimal places

Here's a sample run of the program:

```
Enter a temperature in degrees F: 72
72 degrees F = 22.22 degrees C

Enter a temperature in degrees C: 37
37 degrees C = 98.60 degrees F
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

6.4 Run in Circles

One of the great things about computers is that you can make them do the same thing over and over again, and they never complain!

A **loop** is a block of code that repeatedly executes either a specified number of times or until some condition is met. Python has two kinds of loops: **while loops** and **for loops**. In this section, you'll learn how to use both.

The **while** Loop

`while` loops repeat a section of code while some condition is true. There are two parts to every `while` loop:

1. The **while statement** starts with the `while` keyword, followed by a **test condition**, and ends with a colon (:).
2. The **loop body** contains the code that gets repeated at each step of the loop. Each line is indented four spaces.

When Python executes a `while` loop, it evaluates the test condition and determines if it is true or false. If the test condition is true, then Python executes the code in the loop body and returns to check the test condition again. Otherwise, it skips the code in the body and executes the rest of the program.

Let's look at an example. Type the following code into the interactive window:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...     n = n + 1
...
1
2
3
4
```

First, the integer `1` is assigned to the variable `n`. Then a `while` loop is created with the test condition `n < 5`, which checks whether the value of `n` is less than `5`.

If `n` is less than `5`, then the body of the loop is executed. There are two lines of code in the loop body. In the first line, the value of `n` is printed on the screen. In the second line, `n` is **incremented** by `1`.

The loop execution takes place in five steps:

Step #	Value of n	Test Condition	What Happens
1	1	<code>1 < 5</code> (true)	Prints 1; n increments to 2
2	2	<code>2 < 5</code> (true)	Prints 2; n increments to 3
3	3	<code>3 < 5</code> (true)	Prints 3; n increments to 4
4	4	<code>4 < 5</code> (true)	Prints 4; n increments to 5
5	5	<code>5 < 5</code> (false)	Nothing; loop ends

If you aren't careful, you can create an **infinite loop**. This happens when the test condition is always true. An infinite loop never terminates. The loop body keeps repeating forever.

Here's an example of an infinite loop:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...
```

The only difference between this `while` loop and the previous one is that `n` is never incremented in the loop body. At each step of the loop, `n` is equal to 1. That means the test condition `n < 5` will always be true, and the number 1 will print over and over again forever.

Note

Infinite loops aren't inherently bad. Sometimes they're exactly the kind of loop you need.

For example, code that interacts with hardware may use an infinite loop to constantly check whether a button or switch has been activated.

If you run a program that enters an infinite loop, you can force Python to quit by pressing `Ctrl`+`C`. Python stops running the program and raises a `KeyboardInterrupt` error:

```
Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    print(n)
KeyboardInterrupt
```

Let's look at an example of a `while` loop in practice. One use for a `while` loop is to check whether a user input meets some condition and, if not, repeatedly ask the user for new input until valid input is received.

For instance, the following program continuously asks a user for a positive number until a positive number is entered:

```
num = float(input("Enter a positive number: "))

while num <= 0:
    print("That's not a positive number!")
    num = float(input("Enter a positive number: "))
```

First, the user is prompted to enter a positive number. The test condition `num <= 0` determines whether `num` is less than or equal to 0.

If `num` is positive, then the test condition fails. The body of the loop is skipped and the program ends.

Otherwise, if `num` is 0 or negative, then the body of the loop executes. The program notifies the user that their input was incorrect and prompts them again to enter a positive number.

`while` loops are perfect for repeating a section of code while some condition is met. They aren't well suited, however, for repeating a section of code a specific number of times.

The `for` Loop

A `for` loop executes a section of code once for each item in a collection of items. The number of times that the code is executed is determined by the number of items in the collection.

Like its `while` loop counterpart, the `for` loop has two main parts:

1. The **for statement** begins with the `for` keyword, followed by a **membership expression**, and ends in a colon (`:`).
2. The **loop body** contains the code to be executed at each step of the loop. Each line is indented four spaces.

Let's look at an example. The following `for` loop prints each letter of the string "Python" one at a time:

```
for letter in "Python":  
    print(letter)
```

In this example, the `for` statement is `for letter in "Python"`. The membership expression is `letter in "Python"`.

At each step of the loop, the variable `letter` is assigned the next letter in the string "Python", and then the value of `letter` is printed.

The loop runs once for each character in the string "Python", so the loop body executes six times. The following table summarizes the execution of this `for` loop:

Step #	Value of letter	What Happens
1	"P"	Prints P
2	"y"	Prints y
3	"t"	Prints t
4	"h"	Prints h
5	"o"	Prints o
6	"n"	Prints n

To see why `for` loops are better for looping over collections of items, let's rewrite the `for` loop in the previous example as a `while` loop.

To do so, we can use a variable to store the index of the next character in the string. At each step of the loop, we'll print out the character at the current index and then increment the index.

The loop will stop once the value of the index variable is equal to the length of the string. Remember, indices start at 0, so the last index of the string "Python" is 5.

Here's how you might write that code:

```
word = "Python"  
index = 0  
  
while index < len(word):  
    print(word[index])  
    index = index + 1
```

That's significantly more complex than the `for` loop version!

Not only is the `for` loop less complex, but the code itself looks more natural. It more closely resembles how you might describe the loop in English.

Note

You may sometimes hear people describe some code as being particularly “Pythonic.” The term **Pythonic** is generally used to describe code that is clear, concise, and uses Python’s built-in features to its advantage.

Accordingly, using a `for` loop to loop over a collection of items is more Pythonic than using a `while` loop.

Sometimes it's useful to loop over a range of numbers. Python has a handy built-in function, `range()`, that produces just that—a range of numbers!

For example, `range(3)` returns the range of integers from 0 up to but not including 3. That is, `range(3)` is the range of numbers 0, 1, and 2.

You can use `range(n)`, where `n` is any positive number, to execute a loop exactly `n` times. For instance, the following `for` loop prints the string "Python" three times:

```
for n in range(3):  
    print("Python")
```

You can also give a range a starting point. For example, `range(1, 5)` is the range of numbers 1, 2, 3, and 4. The first argument is the starting number, and the second argument is the endpoint, which is not included in the range.

Using the two-argument version of `range()`, the following `for` loop prints the square of every number from 10 up to but not including 20:

```
for n in range(10, 20):
    print(n * n)
```

Let's look at a practical example. The following program asks the user to input an amount and then displays how to split that amount between two, three, four, and five people:

```
amount = float(input("Enter an amount: "))

for num_people in range(2, 6):
    print(f"{num_people} people: ${amount / num_people:.2f} each")
```

The `for` loop loops over the numbers 2, 3, 4, and 5 and prints the number of people and the amount each person should pay. The formatting specifier `.2f` formats the amount as a fixed-point number rounded to two decimal places and with thousands grouped by commas.

Running the program with the input 10 produces the following output:

```
Enter an amount: 10
2 people: $5.00 each
3 people: $3.33 each
4 people: $2.50 each
5 people: $2.00 each
```

`for` loops are generally used more often in Python than `while` loops. Most of the time, a `for` loop is more concise and easier to read than an equivalent `while` loop.

Nested Loops

As long as you indent the code correctly, you can even put loops inside other loops.

Type the following into IDLE's interactive window:

```
for n in range(1, 4):
    for j in range(4, 7):
        print(f'n = {n} and j = {j}')
```

When Python executes the first `for` loop, it assigns the value 1 to the variable `n`. Then it executes the second `for` loop and assigns the value 4 to `j`. The first result printed is `n = 1 and j = 4`.

After executing `print()`, Python returns to the *inner* `for` loop, assigns 5 to `j`, and prints `n = 1 and j = 5`. Python doesn't return to the outer `for` loop because the inner `for` loop, which is inside the body of the outer `for` loop, isn't done executing.

Next, Python assigns 6 to `j` and prints `n = 1 and j = 6`. At this point, the inner `for` loop is done executing, so control returns to the outer `for` loop. Python assigns the value 2 to the variable `n`, and then the inner `for` loop executes a second time. This process repeats a third time when `n` is assigned the value 3.

The final output looks like this:

```
n = 1 and j = 4
n = 1 and j = 5
n = 1 and j = 6
n = 2 and j = 4
n = 2 and j = 5
n = 2 and j = 6
n = 3 and j = 4
n = 3 and j = 5
n = 3 and j = 6
```

A loop inside another loop is called a **nested loop**, and it comes up

more often than you might expect. You can nest `while` loops inside `for` loops and vice versa. You can even nest loops more than two levels deep!

Important

Nesting loops inherently increases the complexity of your code. You can see that illustrated by the dramatic increase in steps run in the above example as compared to the previous examples with a single `for` loop.

Using nested loops is sometimes the only way to get something done, but too many nested loops can have a negative effect on a program's performance.

Loops are a powerful tool. They tap into one of the greatest advantages that computers provide as tools for computation: the ability to repeat the same task a vast number of times without tiring and without complaining.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a `for` loop that prints out the integers 2 through 10, each on a new line, using `range()`.
2. Write a `while` loop that prints out the integers 2 through 10. (*Hint:* You'll need to create a new integer first.)
3. Write a function called `doubles()` that takes a number as its input and doubles it. Then use `doubles()` in a loop to double the number 2 three times, displaying each result on a separate line. Here's some sample output:

4

8

16

6.5 Challenge: Track Your Investments

In this challenge, you'll write a program called `invest.py` that tracks the growing amount of an investment over time.

The initial deposit for an investment is called the principal amount. Each year, the amount increases by a fixed percentage, called the annual rate of return.

For example, a principal amount of \$100.00 with an annual rate of return of 5 percent increases the first year by \$5.00 for a new amount of \$105.00. The second year, the increase is 5 percent of \$105.00, or \$5.25, bringing the total to \$110.25.

Write a function called `invest` with three parameters: the principal amount, the annual rate of return, and the number of years to calculate. The function signature might look something like this:

```
def invest(amount, rate, years):
```

The function should then print out the amount of the investment, rounded to two decimal places, at the end of each year for the specified number of years.

For example, calling `invest(100, .05, 4)` should print the following:

```
year 1: $105.00
year 2: $110.25
year 3: $115.76
year 4: $121.55
```

To finish the program, prompt the user to enter an initial amount, an annual percentage rate, and a number of years. Then call `invest()` to display the calculations for the values entered by the user.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

6.6 Understand Scope in Python

Any discussion of functions and loops in Python would be incomplete without some mention of the issue of **scope**.

Scope can be one of the more difficult programming concepts to understand, so in this section you'll get a gentle introduction to it.

By the end of this section, you'll know what a **scope** is and why it's important. You'll also learn the LEGB rule for **scope resolution**.

What Is a Scope?

When you assign a value to a variable, you're giving that value a name. Names are unique. For example, you can't assign the same name to two different numbers:

```
>>> x = 2
>>> x
2

>>> x = 3
>>> x
3
```

When you assign 3 to x, you can no longer recall the value 2 with the name x.

This behavior makes sense. After all, if the variable x had the values 2 and 3 simultaneously, then how would you evaluate $x + 2$? Would it be 4 or 5?

As it turns out, there *is* a way to assign the same name to two different values. It just involves a bit of trickery.

Open a new editor window in IDLE and type out the following program:

```
x = "Hello, World"

def func():
    x = 2
    print(f"Inside 'func', x has the value {x}")

func()
print(f"Outside 'func', x has the value {x}")
```

In this example, you assign two different values to the variable `x`: "Hello, World" at the beginning and 2 inside `func()`.

The code's output, which you might find surprising, looks like this:

```
Inside 'func', x has the value 2
Outside 'func', x has the value Hello, World
```

How does `x` still have the value "Hello, World" after calling `func()`, which changes the value of `x` to 2?

The answer is that `func()` has a different **scope** than the code that exists outside the function. That is, you can give an object inside `func()` the same name as an object outside `func()`, and Python can keep the two separated.

The function body has what's known as a **local scope**, with its own set of names available to it. Code outside the function body is in the **global scope**.

You can think of a scope as a set of names mapped to objects. When you use a particular name in your code, such as a variable or a function name, Python checks the current scope to determine whether that name exists.

Scope Resolution

Scopes have a hierarchy. For example, consider the following:

```
x = 5

def outer_func():
    y = 3

    def inner_func():
        z = x + y
        return z

    return inner_func()
```

Note

`inner_func()` is called an **inner function** because it's defined inside another function. Just like you can nest loops, you can define functions within other functions!

You can read more about inner functions in Real Python's article "[Inner Functions—What Are They Good For?](#)"

The variable `z` is in the local scope of `inner_func()`. When Python executes the line `z = x + y`, it looks for the variables `x` and `y` in the local scope. Neither of them exist there, so Python moves up to the scope of `outer_func()`.

The scope for `outer_func()` is an **enclosing** scope of `inner_func()`. It's not quite the global scope, nor is it the local scope for `inner_func()`. It lies between the two.

The variable `y` is defined in the scope for `outer_func()` and is assigned the value 3. However, `x` does not exist in this scope, so Python moves up once again to the global scope. There it finds the name `x`, which has the value 5. Now that the names `x` and `y` are resolved, Python can execute the line `z = x + y`, which assigns to `z` the value of 8.

The LEGB Rule

A useful way to remember how Python resolves scope is with the **LEGB** rule. LEGB is an initialism for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in, which describes the order by which Python resolves scope.

Here's a quick overview to help you remember how all of this works:

1. **Local:** The local, or current, scope could be the body of a function or the top-level scope of a code file. It always represents the scope that the Python interpreter is currently working in.
2. **Enclosing:** The enclosing scope is the scope one level up from the local scope. If the local scope is an inner function, then the enclosing scope is the scope of the outer function. If the scope is a top-level function, then the enclosing scope is the same as the global scope.
3. **Global:** The global scope is the topmost scope in a program. It contains all the names defined in the code that are not contained in a function body.
4. **Built-in:** The built-in scope contains all the names, such as keywords, that are built into Python. Functions such as `round()` and `abs()` are in the built-in scope. Anything that you can use without first defining it yourself is contained in the built-in scope.

Scope can be confusing, and it takes some practice for the concept to feel natural. Don't worry if it doesn't make sense at first. Just keep practicing and use the LEGB rule to help you figure things out.

Break the Rules

Consider what the output would be for the following code:

```
total = 0

def add_to_total(n):
    total = total + n

add_to_total(5)
print(total)
```

You would think that the program outputs the value 5, right? Try running it to see what happens.

Something unexpected occurs! You get an error:

```
Traceback (most recent call last):
  File "C:/Users/davea/stuff/python/scope.py", line 6, in <module>
    add_to_total(5)
  File "C:/Users/davea/stuff/python/scope.py", line 4, in add_to_total
    total = total + n
UnboundLocalError: local variable 'total' referenced before assignment
```

Wait a minute! According to the LEGB rule, Python should have recognized that the name `total` doesn't exist in the `add_to_total()` function's local scope and then moved up to the global scope to resolve the name, right?

The problem here is that the code attempts to make an assignment to the variable `total`, which creates a new name in the local scope. Then, when Python executes the right-hand side of the assignment, it finds the name `total` in the local scope with nothing assigned to it yet.

These kinds of errors are tricky and are one of the reasons it's best to use unique variable and function names no matter which scope you're in.

You can get around this issue with the `global` keyword:

```
total = 0

def add_to_total(n):
    global total
    total = total + n

add_to_total(5)
print(total)
```

This time, you get the expected output 5. Why's that?

The line `global total` tells Python to look in the global scope for the name `total`. That way, the line `total = total + n` doesn't create a new local variable.

Although this “fixes” the program, the use of the `global` keyword is considered bad form in general.

If you find yourself using `global` to fix problems like the one above, stop and think if there's a better way to write your code. Often, you'll find that there is!

6.7 Summary and Additional Resources

In this chapter, you learned about two of the most essential concepts in programming: functions and loops.

First, you learned how to define your own custom functions. You saw that functions are made up of two parts:

1. The **function signature**, which starts with the `def` keyword and includes the name of the function and the function's parameters
2. The **function body**, which contains the code that runs whenever the function is called

Functions help you avoid repeating similar code throughout your program by creating re-usable components. This helps make your code easier to read and maintain.

Then you learned about Python's two kinds of loops:

1. **while loops** repeat some code while a given condition remains true.
2. **for loops** repeat some code for each element in a set of objects.

Finally, you learned what a **scope** is and how Python resolves scope using the LEGB rule.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-functions-loops

Additional Resources

To learn more about functions and loops, check out the following resources:

- “Python ‘while’ Loops (Indefinite Iteration)”
- “Python ‘for’ Loops (Definite Iteration)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 7

Finding and Fixing Code Bugs

Everyone makes mistakes—even seasoned professional developers!

IDLE is pretty good at catching mistakes like syntax errors and run-time errors, but there's a third type of error that you may have already experienced. **Logic errors** occur when an otherwise valid program doesn't do what was intended.

Logic errors cause unexpected behaviors called **bugs**. Removing bugs is called **debugging**, and a **debugger** is a tool that helps you hunt down bugs and understand why they're happening.

Knowing how to find and fix bugs in your code is a skill that you will use for your entire coding career!

In this chapter, you will:

- Learn how to use IDLE's Debug Control window
- Practice debugging on a buggy function

Let's go!

7.1 Use the Debug Control Window

The main interface to IDLE’s debugger is the Debug Control window, which we’ll refer to as the Debug window for short. You can open the Debug window by selecting `Debug > Debugger` from the menu in the interactive window. Go ahead and open the Debug window.

Important

If the `Debug` menu is missing from your menu bar, make sure to bring the interactive window into focus by clicking it.

Whenever the Debug window is open, the interactive window displays `[DEBUG ON]` next to the prompt to indicate that the debugger is open. Now open a new editor window and arrange the three windows on your screen so that you can see all of them simultaneously.

In this section, you’ll learn how the Debug window is organized, how to step through your code with the debugger one line at a time, and how to set breakpoints to help speed up the debugging process.

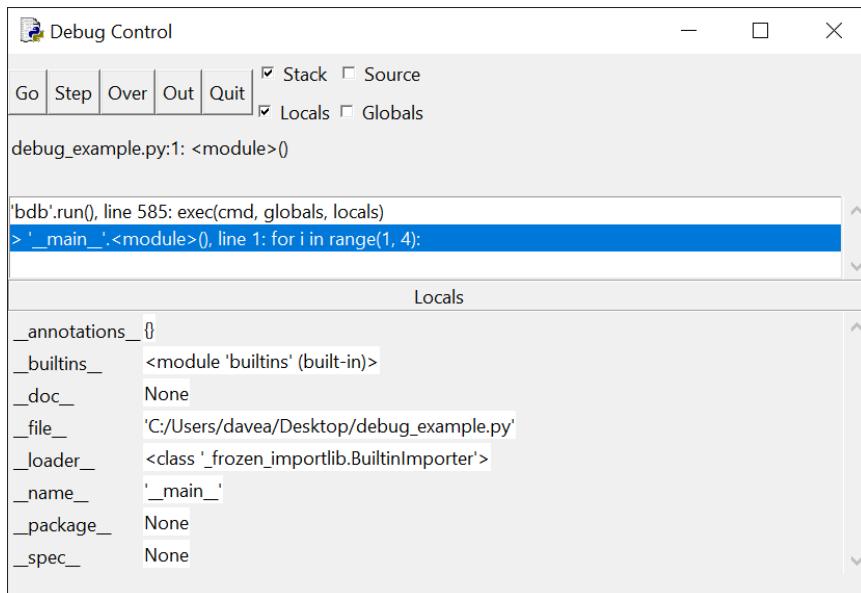
The Debug Control Window: An Overview

To see how the debugger works, let’s start by writing a simple program without any bugs. Type the following into the editor window:

```
for i in range(1, 4):
    j = i * 2
    print(f"i is {i} and j is {j}")
```

Save the file, then keep the Debug window open and press `F5`. You’ll notice that execution doesn’t get very far.

The Debug window will look like this:



Notice that the Stack panel at the top of the window contains the following message:

```
> '__main__'.<module>(), line 1: for i in range(1, 4):
```

This tells you that line 1 (which contains the code `for i in range(1, 4):`) is *about* to be run but hasn't started yet. The `'__main__'.module()` part of the message refers to the fact that you're currently in the main section of the program, as opposed to being, for example, in a function definition before the main block of code has been reached.

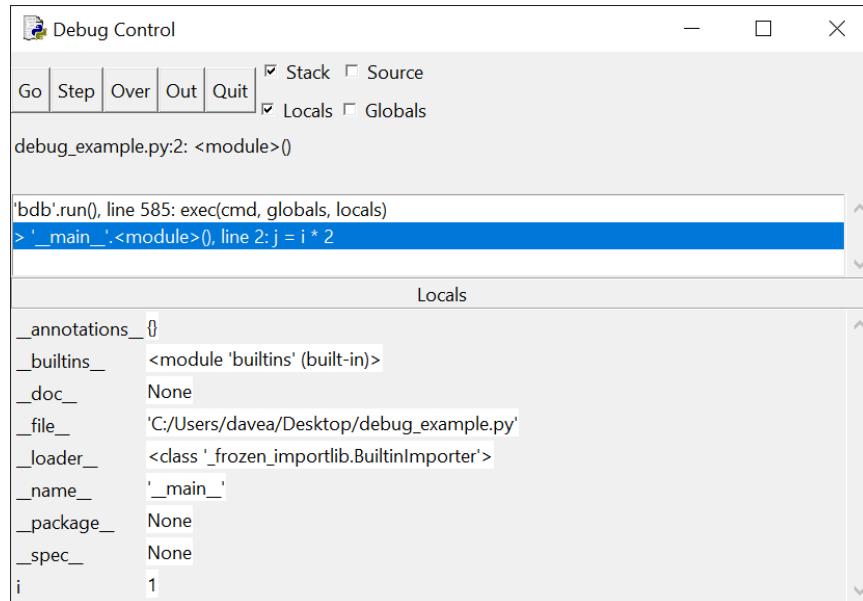
Below the Stack panel is a Locals panel that lists some strange looking stuff like `__annotations__`, `__builtins__`, `__doc__`, and so on. These are internal system variables that you can ignore for now. As your program runs, you'll see variables declared in the code displayed in this window so that you can keep track of their value.

There are five buttons located at the top left-hand corner of the Debug window: **Go**, **Step**, **Over**, **Out**, and **Quit**. These buttons control how the debugger moves through your code.

In the following sections, we'll explore what each of these buttons does, starting with **Step**.

The Step Button

Go ahead and click **Step** at the top left-hand corner of the Debug window. The Debug window changes a bit to look like this:



There are two differences to pay attention to here. First, the message in the Stack panel changes to the following:

```
> '__main__'.<module>(), line 2: j = i * 2:
```

At this point, line 1 of your code has run, and the debugger has stopped just before executing line 2.

The second change to notice is the new variable `i` that is assigned the value `1` in the Locals panel. That's because the `for` loop in the first line of code created the variable `i` and assigned it the value `1`.

Continue hitting the `Step` button to walk through your code line by line, and watch what happens in the debugger window. When you arrive at the line `print(f"i is {i} and j is {j}")`, you can see the output displayed in the interactive window one piece at a time.

More importantly, you can track the growing values of `i` and `j` as you step through the `for` loop. You can probably imagine how beneficial this feature is when trying to locate the source of bugs in your programs. Knowing each variable's value at each line of code can help you pinpoint where things go wrong.

Breakpoints and the Go Button

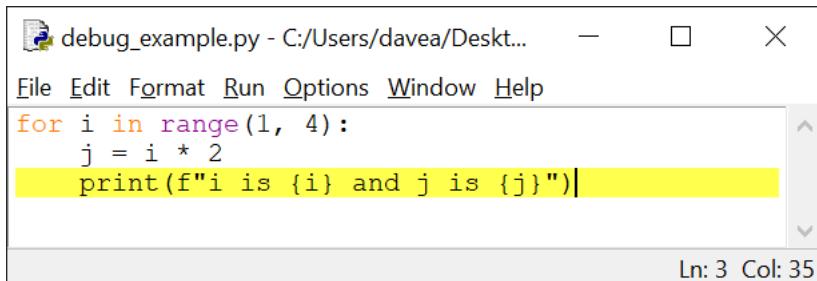
Often, you may know that the bug must be in a particular section of your code, but you may not know precisely where. Rather than clicking the `Step` button all day long, you can set a **breakpoint** that tells the debugger to continuously run all code until it reaches the breakpoint.

Breakpoints tell the debugger when to pause code execution so that you can take a look at the current state of the program. They don't actually break anything.

To set a breakpoint, right-click (`Ctrl`-click on a Mac) the line of code in your editor window that you would like to pause at and select `Set Breakpoint`. IDLE highlights the line in yellow to indicate that your breakpoint has been set. To remove a breakpoint, right-click the line with the breakpoint and select `Clear Breakpoint`.

Go ahead and press `Quit` at the top of the Debug window to turn off the debugger for now. This won't close the window, and you'll want to keep it open because you'll be using it again in just a moment.

Set a breakpoint on the line of code with the `print()` function call. The editor window should now look like this:

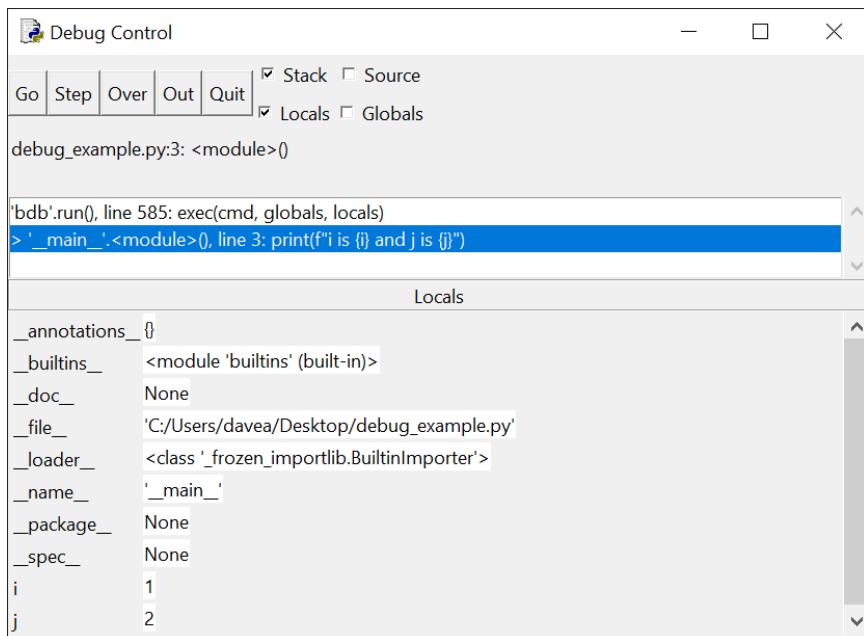


A screenshot of a Python code editor window titled "debug_example.py - C:/Users/davea/Desktop...". The code is as follows:

```
for i in range(1, 4):
    j = i * 2
    print(f'i is {i} and j is {j}')
```

The line `print(f'i is {i} and j is {j}')` is highlighted with a yellow background, indicating it is the current line of execution.

Save and run the file. Just like before, the Stack panel of the Debug window indicates that the debugger has started and is waiting to execute line 1. Click `Go` and watch what happens in the Debug window:



A screenshot of the "Debug Control" window. The toolbar includes buttons for Go, Step, Over, Out, and Quit, with "Go" being the active button. There are also checkboxes for Stack (checked), Source (unchecked), Locals (checked), and Globals (unchecked). The status bar shows "debug_example.py:3: <module>()".

The main pane shows the stack trace:

```
'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__.<module>()', line 3: print(f'i is {i} and j is {j}')
```

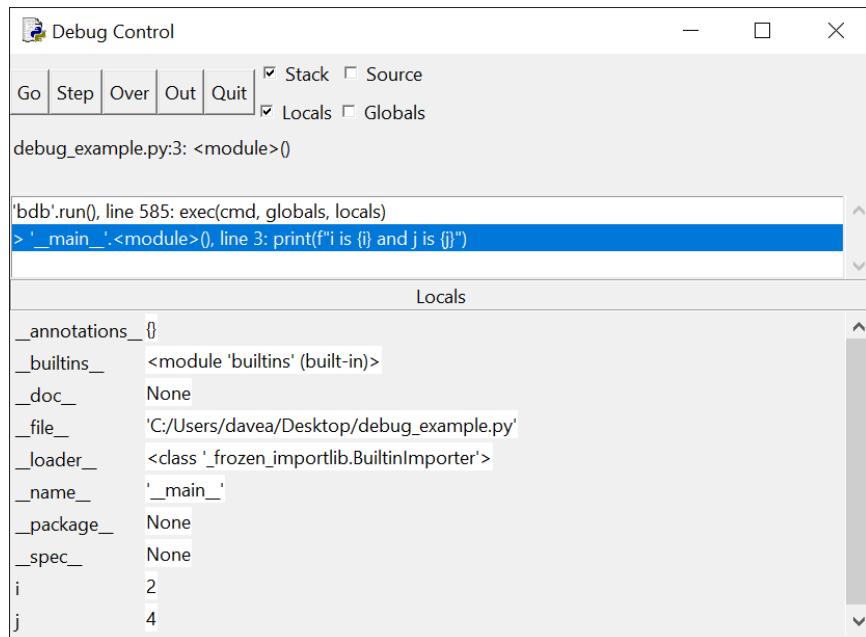
Below the stack trace is a "Locals" table:

Locals	
<code>_annotations_</code>	0
<code>_builtins_</code>	<module 'builtins' (built-in)>
<code>_doc_</code>	None
<code>_file_</code>	'C:/Users/davea/Desktop/debug_example.py'
<code>_loader_</code>	<class '_frozen_importlib.BuiltinImporter'>
<code>_name_</code>	'__main__'
<code>_package_</code>	None
<code>_spec_</code>	None
<code>i</code>	1
<code>j</code>	2

The Stack panel now shows the following message indicating that it's waiting to execute line 3:

```
> '__main__'.<module>(), line 3: print(f'i is {i} and j is {j}')
```

If you look at the Locals panel, then you'll see that both variables `i` and `j` have the values `1` and `2`, respectively. By clicking `[Go]`, you told the debugger to run your code continuously until it reached either a breakpoint or the end of the program. Press `[Go]` again. The Debug window now looks like this:



Do you see what changed? The same message as before is displayed in the Stack panel, indicating that the debugger is waiting to execute line 3 again. However, the values of the variables `i` and `j` are now `2` and `4`. The interactive window also displays the output from having run the line with `print()` the first time through the loop.

Each time you press the `[Go]` button, the debugger runs the code continuously until it reaches the next breakpoint. Since you set the break-

point on line 3, which is inside the `for` loop, the debugger stops on this line each time it goes through the loop.

Press `Go` a third time. Now `i` and `j` have the values 3 and 6. What do you think will happen when you press `Go` one more time? Since the `for` loop only iterates three times, when you press `Go` again, the program will finish running.

Over and Out

The `Over` button works sort of like a combination of `Step` and `Go`. It steps over a function or a loop. In other words, if you're about to `Step` into a function with the debugger, then you can still run that function's code without having to `Step` all the way through each line of it. The `Over` button takes you directly to the result of running that function.

Likewise, if you're already inside a function or loop, then the `Out` button executes the remaining code inside the function or loop body and then pauses.

In the next section, you'll look at some buggy code and learn how to fix it with IDLE.

7.2 Squash Some Bugs

Now that you've gotten comfortable with using the Debug Control window, let's take a look at a buggy program.

The following code defines a function `add_underscores()` that takes a single string object `word` as an argument and returns a new string containing a copy of `word` with each character surrounded by underscores. For example, `add_underscores("python")` should return `"_p_y_t_h_o_n_"`.

Here's the buggy code:

```
def add_underscores(word):
    new_word = "_"
    for i in range(len(word)):
        new_word = word[i] + "_"
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

Type this code into the editor window, then save the file and press **F5** to run the program. The expected output is `_h_e_l_l_o_`, but instead all you see is `o_`, or the letter "o" followed by a single underscore.

If you already see what the problem with the code is, don't just fix it. The point of this section is to learn how to use IDLE's debugger to identify the problem.

If you don't see what the problem is, don't worry! By the end of this section, you'll have found it and will be able to identify similar problems in other code you encounter.

Note

Debugging can be difficult and time consuming, and bugs can be subtle and hard to identify.

While this section looks at a relatively simple bug, the method used to inspect the code and find the bug is the same for more complex problems.

Debugging is problem solving, and as you become more experienced, you'll develop your own approaches. In this section, you'll learn a simple four-step method to help get you started:

1. Guess which section of code may contain the bug.
2. Set a breakpoint and inspect the code by stepping through the

buggy section one line at a time, keeping track of important variables along the way.

3. Identify the line of code, if any, with the error and make a change to solve the problem.
4. Repeat steps 1–3 as needed until the code works as expected.

Step 1: Make a Guess About Where the Bug Is Located

The first step is to identify the section of code that likely contains the bug. You may not be able to identify exactly where the bug is at first, but you can usually make a reasonable guess about which section of your code has an error.

Notice that the program is split into two distinct sections: a function definition (where `add_underscores()` is defined), and a main code block that defines a variable `phrase` with the value "hello" and then prints the result of calling `add_underscores(phrase)`.

Look at the main section:

```
phrase = "hello"  
print(add_underscores(phrase))
```

Do you think the problem could be here? It doesn't look like it, right? Everything about those two lines of code looks good. So, the problem must be in the function definition:

```
def add_underscores(word):  
    new_word = "_"  
    for i in range(len(word)):  
        new_word = word[i] + "_"  
    return new_word
```

The first line of code inside the function creates a variable `new_word` with the value `_`. You're all good there, so you can conclude that the problem is somewhere in the body of the `for` loop.

Step 2: Set a Breakpoint and Inspect the Code

Now that you've identified where the bug must be, set a breakpoint at the start of the `for` loop so that you can trace out exactly what's happening inside the code with the Debug window:

```
squash_some_bugs.py - C:/Users/davea/Desktop/squash_so...
File Edit Format Run Options Window Help
def add_underscores(word):
    new_word = " "
    for i in range(0, len(word)):
        new_word = word[i] + "_"
    return new_word

phrase = "hello "
print(add_underscores(phrase))

Ln: 3 Col: 33
```

Open the Debug window and run the file. Execution still pauses on the very first line it sees, which is the function definition.

Press `Go` to run through the code until the breakpoint is encountered. The Debug window will now look like this:

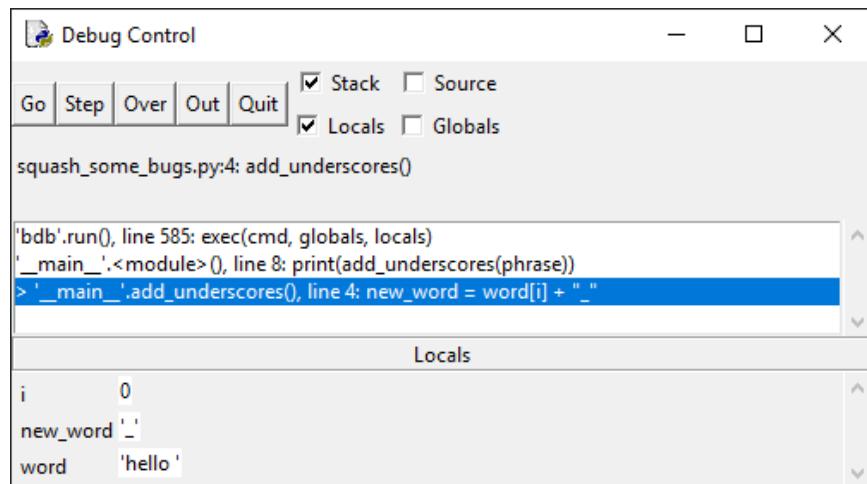
```
Debug Control
Go Step Over Out Quit Stack Source
Locals Globals
squash_some_bugs.py:3: add_underscores()

'bdb'.run(), line 585: exec(cmd, globals, locals)
'_main_'.<module>(), line 8: print(add_underscores(phrase))
> '_main_'.add_underscores(), line 3: for i in range(0, len(word)):

Locals
new_word '_'
word     'hello '
```

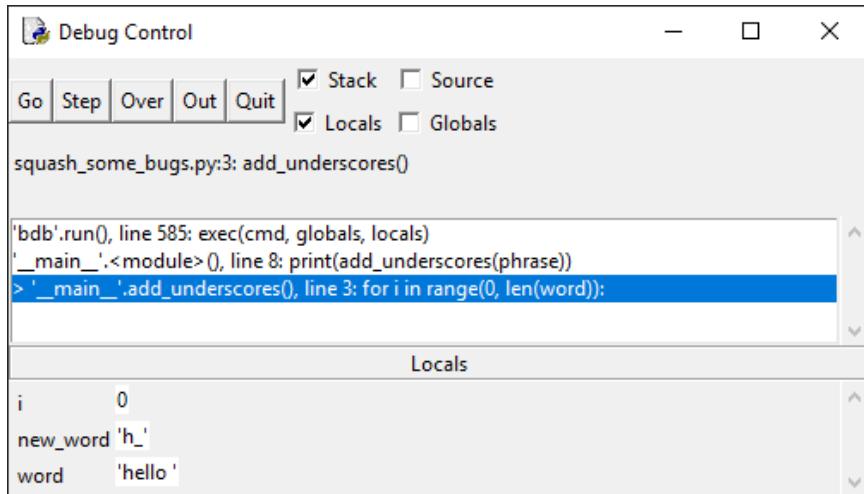
At this point, the code is paused just before entering the `for` loop in the `add_underscores()` function. Notice that two local variables, `word` and `new_word`, are displayed in the Locals panel. Currently, `word` has the value "`hello`" and `new_word` has the value "`_`" as expected.

Click `Step` once to enter the `for` loop. The Debug window changes, and a new variable `i` with the value `0` is displayed in the Locals panel:



`i` is the counter used in the `for` loop, and you can use it to keep track of which iteration of the `for` loop you're currently looking at.

Click **Step** one more time. If you look at the Locals panel, then you'll see that the variable `new_word` has taken on the value "`h_`":



This isn't right. Originally, `new_word` had the value `"_"`, and on the second iteration of the `for` loop it should now have the value `"_h_"`. If you click **Step** a few more times, then you'll see that `new_word` gets set to `e_`, then `l_`, and so on.

Step 3: Identify the Error and Attempt to Fix It

The conclusion you can make at this point is that, at each iteration of the `for` loop, `new_word` is overwritten with the next character in the string `"hello"` and a trailing underscore. Since there's only one line of code inside the `for` loop, you know that the problem must be with the following code:

```
new_word = word[i] + "_"
```

Look at the line closely. It tells Python to get the next character of `word`, tack an underscore onto the end of it, and assign this new string to the variable `new_word`. This is exactly the behavior you've witnessed by stepping through the `for` loop!

To fix the problem, you need to tell Python to concatenate the string `word[i] + "_"` to the existing value of `new_word`. Press `Quit` in the Debug window, but don't close the window just yet. Open the editor window and change the line inside the `for` loop to the following:

```
new_word = new_word + word[i] + "_"
```

Step 4: Repeat Steps 1–3 Until the Bug Is Gone

Save the new changes to the program and run it again. In the Debug window, press `Go` to execute the code up to the breakpoint.

Note

If you closed the debugger in the previous step without clicking `Quit`, then you may see the following error when reopening the Debug window:

You can only toggle the debugger when idle

Always be sure to click `Go` or `Quit` when you're finished with a debugging session instead of just closing the debugger, or you might have trouble reopening it. To get rid of this error, you'll have to close and reopen IDLE.

The program pauses just before entering the `for` loop in `add_underscores()`. Press `Step` repeatedly and watch what happens to the `new_word` variable at each iteration. Success! Everything works as expected!

Your first attempt at fixing the bug worked, so you don't need to repeat steps 1–3 anymore. This won't always be the case. Sometimes you'll have to repeat the process several times before you fix a bug.

Alternative Ways to Find Bugs

Using a debugger can be tricky and time consuming, but it's the most reliable way to find bugs in your code. Debuggers aren't always avail-

able, though. Systems with limited resources, such as small Internet of Things devices, often won't have built-in debuggers.

In situations like these, you can use **print debugging** to find bugs in your code. Print debugging uses `print()` to display text in the console that indicates where the program is executing and what the state of the program's variables are at certain points in the code.

For example, instead of debugging the previous program with the Debug window, you could add the following line to the end of the `for` loop in `add_underscores()`:

```
print(f"i = {i}; new_word = {new_word}")
```

The altered code would then look like this:

```
def add_underscores(word):
    new_word = "_"
    for i in range(len(word)):
        new_word = word[i] + "_"
        print(f"i = {i}; new_word = {new_word}")
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

When you run the file, the interactive window displays the following output:

```
i = 0; new_word = h_
i = 1; new_word = e_
i = 2; new_word = l_
i = 3; new_word = l_
i = 4; new_word = o_
o_
```

This shows you what the value of `new_word` is at each iteration of the `for` loop. The final line containing just a single underscore is the result of running `print(add_underscore(phrase))` at the end of the program.

By looking at the above output, you can come to the same conclusion you did while debugging with the Debug window. The problem is that `new_word` is overwritten at each iteration.

Print debugging works, but it has several disadvantages over debugging with a debugger. First, you have to run your entire program each time you want to inspect the values of your variables. This can be an enormous waste of time compared to using breakpoints. You also have to remember to remove those `print()` function calls from your code when you're done debugging!

The example loop in this section may be a good example for illustrating the process of debugging, but it's not the best example of Pythonic code. The use of the index `i` is a giveaway that there might be a better way to write the loop.

One way to improve this loop is to iterate over the characters in `word` directly. Here's one way to do that:

```
def add_underscores(word):
    new_word = "_"_
    for letter in word:
        new_word = new_word + letter + "_"
    return new_word
```

The process of rewriting existing code to be cleaner, easier to read and understand, or more in line with the standards set by a team is called **refactoring**. We won't discuss refactoring much in this book, but it's an essential part of writing professional-quality code.

7.3 Summary and Additional Resources

In this chapter, you learned about IDLE's Debug window. You saw how to inspect the values of variables, insert breakpoints, and use the `Step`, `Go`, `Over`, and `Out` buttons.

You also got some practice debugging a faulty function using a four-step process for identifying and removing bugs:

1. Guess where the bug is located.
2. Set a breakpoint and inspect the code.
3. Identify the error and attempt to fix it.
4. Repeat steps 1–3 until the error is fixed.

Debugging is as much an art as it is a science. The only way to master debugging is to get a lot of practice with it!

One way to get some practice is to open the Debug Control window and use it to step through your code as you work on the exercises and challenges throughout the rest of this book.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-debugging

Additional Resources

For more information on debugging Python code, check out the following resources:

- “Python Debugging With `pdb`” (Article)
- “Python Debugging With `pdb`” (Video Course)

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 8

Conditional Logic and Control Flow

Nearly all of the code you've seen in this book is **unconditional**. That is, the code does not make any choices. Every line of code is executed in the order that it's written or in the order that functions are called, with possible repetitions inside loops.

In this chapter, you'll learn how to use **conditional logic** to write programs that perform different actions based on different conditions. Paired with functions and loops, conditional logic allows you to write complex programs that can handle many different situations.

In this chapter, you'll learn how to:

- Compare the values of two or more variables
- Write `if` statements to control the flow of your programs
- Handle errors with `try` and `except`
- Apply conditional logic to create simple simulations

Let's get started!

8.1 Compare Values

Conditional logic is based on performing different actions depending on whether an expression, called a **conditional**, is true or false. This idea is not specific to computers. Humans use conditional logic all the time to make decisions.

For example, the legal age for purchasing alcoholic beverages in the United States is twenty-one. The statement “If you are at least twenty-one years old, then you may purchase a beer” is an example of conditional logic. The clause “If you are at least twenty-one years old” is a conditional because it establishes a condition—that you are age twenty-one or older—which may be either true or false.

In computer programming, conditionals often take the form of a comparison between two values, such as determining if one value is greater than another or whether two values are equal. A standard set of symbols called **Boolean comparators** are used to make comparisons, and most of them may be familiar to you already.

The following table describes these Boolean comparators:

Boolean Comparator	Example	Meaning
>	$a > b$	a greater than b
<	$a < b$	a less than b
\geq	$a \geq b$	a greater than or equal to b
\leq	$a \leq b$	a less than or equal to b
\neq	$a \neq b$	a not equal to b
\equiv	$a \equiv b$	a equal to b

The term **Boolean** is derived from the last name of the English mathematician George Boole, whose works helped lay the foundations of modern computing. In Boole’s honor, conditional logic is sometimes called **Boolean logic**, and conditionals are sometimes called **Boolean expressions**.

There is also a fundamental data type called the **Boolean**, or `bool` for short, that can have only one of two values. In Python, these values are conveniently named `True` and `False`:

```
>>> type(True)
<class 'bool'>

>>> type(False)
<class 'bool'>
```

Note that `True` and `False` both start with capital letters.

The result of evaluating a conditional is always a Boolean value:

```
>>> 1 == 1
True

>>> 3 > 5
False
```

In the first example, since `1` is equal to `1`, the result of `1 == 1` is `True`. In the second example, `3` is not greater than `5`, so the result is `False`.

Important

A common mistake when writing conditionals is to use the assignment operator `=` instead of `==` to test whether two values are equal.

Fortunately, Python will raise a `SyntaxError` if it encounters this mistake, so you'll know about it before you run your program.

You may find it helpful to think of Boolean comparators as asking a question about two values. `a == b` asks whether `a` and `b` have the same value. Likewise, `a != b` asks whether `a` and `b` have different values.

Conditional expressions are not limited to comparing numbers. You can also use them to compare values such as strings:

```
>>> "a" == "a"
```

```
True
```

```
>>> "a" == "b"
```

```
False
```

```
>>> "a" < "b"
```

```
True
```

```
>>> "a" > "b"
```

```
False
```

The last two examples above may look funny to you. How could one string be greater than or less than another?

The comparators `<` and `>` represent the notions of greater than and less than when used with numbers, but more generally they represent the notion of order. In this regard, `"a" < "b"` checks if the string `"a"` comes before the string `"b"`. But how are strings ordered?

In Python, strings are ordered **lexicographically**, which is a fancy way of saying they're ordered as they would appear in a dictionary. So you can think of `"a" < "b"` as asking whether the letter `a` comes before the letter `b` in the dictionary.

Lexicographic ordering extends to strings of two or more characters by looking at each component letter of the string:

```
>>> "apple" < "astronaut"
```

```
True
```

```
>>> "beauty" > "truth"
```

```
False
```

Since strings can contain characters other than letters of the alphabet, the ordering must extend to those other characters as well. Every character has a corresponding number called its **Unicode code point**. When Python compares two characters, it converts the characters to their code points and then compares those two numbers.

We won't go into detail about how Unicode code points work. In practice, the `<` and `>` comparators are most often used with numbers, not strings.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. For each of the following conditional expressions, guess whether they evaluate to `True` or `False`. Then type them into the interactive window to check your answers:
 - `1 <= 1`
 - `1 != 1`
 - `1 != 2`
 - `"good" != "bad"`
 - `"good" != "Good"`
 - `123 == "123"`
2. For each of the following expressions, fill in the blank (indicated by `_`) with an appropriate Boolean comparator so that the expression evaluates to `True`:
 - `3 ___ 4`
 - `10 ___ 5`
 - `"jack" ___ "jill"`
 - `42 ___ "42"`

8.2 Add Some Logic

In addition to Boolean comparators, Python has special keywords called **logical operators** that can be used to combine Boolean expressions. There are three logical operators: `and`, `or`, and `not`.

Logical operators are used to construct compound logical expressions. For the most part, these have meanings similar to their meaning in the English language, although the rules regarding their use in Python are much more precise.

The `and` Keyword

Consider the following statements:

1. Cats have four legs.
2. Cats have tails.

In general, both of these statements are true.

When we combine these two statements using `and`, the resulting sentence “Cats have four legs and cats have tails” is also a true statement. If both statements are negated, then the compound statement “Cats do not have four legs and cats do not have tails” is false.

Even when we mix and match true and false statements, the compound statement is false. “Cats have four legs and cats do not have tails” and “Cats do not have four legs and cats have tails” are both false statements.

When two statements P and Q are combined with `and`, the **truth value** of the compound statement “ P and Q ” is true if and only if both P and Q are true.

Python’s `and` operator works exactly the same way. Here are four examples of compound statements with `and`:

```
>>> 1 < 2 and 3 < 4 # Both are True
```

```
True
```

Both statements are True, so their combination is also True.

```
>>> 2 < 1 and 4 < 3 # Both are False
```

```
False
```

Both statements are False, so their combination is also False.

```
>>> 1 < 2 and 4 < 3 # Second statement is False
```

```
False
```

$1 < 2$ is True, but $4 < 3$ is False, so their combination is False.

```
>>> 2 < 1 and 3 < 4 # First statement is False
```

```
False
```

$2 < 1$ is False, and $3 < 4$ is True, so their combination is False.

The following table summarizes the rules for the `and` operator:

Combination Using <code>and</code>	Result
True and True	True
True and False	False
False and True	False
False and False	False

You can test each of these rules in the interactive window:

```
>>> True and True
```

```
True
```

```
>>> True and False
```

```
False
```

```
>>> False and True
```

```
False  
  
>>> False and False  
False
```

The or Keyword

When we use the word *or* in everyday conversation, sometimes we mean an **exclusive or**. That is, only the first option or the second option can be true.

For example, the statement “I can stay or I can go” uses the exclusive *or*. I can’t both stay and go. Only one of these options can be true.

In Python, however, the *or* keyword is **inclusive**. That is, if *P* and *Q* are two expressions, then the statement “*P* or *Q*” is true if any of the following are true:

1. *P* is true.
2. *Q* is true.
3. Both *P* and *Q* are true.

Let’s look at some examples using numerical comparisons:

```
>>> 1 < 2 or 3 < 4 # Both are True  
True  
  
>>> 2 < 1 or 4 < 3 # Both are False  
False  
  
>>> 1 < 2 or 4 < 3 # Second statement is False  
True  
  
>>> 2 < 1 or 3 < 4 # First statement is False  
True
```

Note that if any part of a compound statement is `True`, even if the other part is `False`, then the result is always `True`.

The following table summarizes the possible combinations with Python's `or` keyword:

Combination Using <code>or</code>	Result
True or True	True
True or False	True
False or True	True
False or False	False

Again, you can verify all of this in the interactive window:

```
>>> True or True
True

>>> True or False
True

>>> False or True
True

>>> False or False
False
```

The `not` Keyword

The `not` keyword reverses the truth value of a single expression:

Use of <code>not</code>	Result
<code>not True</code>	False
<code>not False</code>	True

You can verify this in the interactive window:

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```

One thing to keep in mind with `not`, though, is that it doesn't always behave the way you might expect when combined with comparators like `==`.

For example, `not True == False` returns `True`, but `False == not True` will raise an error:

```
>>> not True == False
```

```
True
```

```
>>> False == not True
```

```
File "<stdin>", line 1
```

```
    False == not True
```

```
^
```

```
SyntaxError: invalid syntax
```

This happens because Python parses logical operators according to an **operator precedence**, just like arithmetic operators have an order of precedence in everyday math.

The order of precedence for logical and Boolean operators, from highest to lowest, is described in the following table. Operators on the same row have equal precedence.

Operator Order of Precedence (Highest to Lowest)

`<, <=, ==, >=, >`

`not`

`and`

`or`

Look again at the expression `False == not True`. Since `not` has a lower precedence than `==` in the order of operations, Python first tries to evaluate `False == not`, which is syntactically incorrect.

You can avoid the `SyntaxError` by surrounding `not True` with parentheses:

```
>>> False == (not True)  
True
```

Grouping expressions with parentheses is a great way to clarify which operators belong to which part of a compound expression. Even if parentheses aren't required, it's a good idea to use them to make compound expressions easier to read.

Building Complex Expressions

You can combine the `and`, `or`, and `not` keywords with `True` and `False` to create more complex expressions. Here's an example of a more complex expression:

```
True and not (1 != 1)
```

What do you think the value of this expression is?

To find out, break the expression down by starting on the right side. `1 != 1` is `False` since `1` has the same value as itself. So you can simplify the above expression as follows:

```
True and not (False)
```

Now, `not (False)` is the same as `not False`, which is `True`. So you can simplify the above expression once more:

```
True and True
```

Finally, `True and True` is just `True`. So, after a few steps, you can see that `True and not (1 != 1)` evaluates to `True`.

When working through complicated expressions, the best strategy is to start with the most complicated part of the expression and build outward from there.

For instance, try evaluating the following expression:

```
("A" != "A") or not (2 >= 3)
```

Start by evaluating the two expressions in parentheses. `"A" != "A"` is `False` because `"A"` is equal to itself. `2 >= 3` is also `False` because 2 is less than 3. This gives you the following equivalent, but simpler, expression:

```
(False) or not (False)
```

Since `not` has a higher precedence than `or`, the above expression is equivalent to the following:

```
False or (not False)
```

`not False` is `True`, so you can simplify the expression even further:

```
False or True
```

Finally, since any compound expression using `or` is `True` when at least one of the expressions on either side of `or` is `True`, you can conclude that `("A" != "A") or not (2 >= 3)` is `True`.

In a compound conditional statement, grouping expressions with parentheses improves readability. Sometimes the parentheses are even required to produce the expected value.

For example, upon first inspection, most people would expect the following to output `True`, but it actually returns `False`:

```
>>> True and False == True and False  
False
```

The reason this is `False` is that the `==` operator has a higher precedence than `and`, so Python interprets the expression as `True and (False ==`

`True`) and `False`. Since `False == True` is `False`, this is equivalent to `True` and `False` and `False`, which evaluates to `False`.

Here's how to add parentheses so that the expression evaluates to `True`:

```
>>> (True and False) == (True and False)
```

```
True
```

Logical operators and Boolean comparators can be confusing the first time you encounter them, so if you don't feel like the material in this section comes naturally, don't worry!

With a little bit of practice, you'll be able to make sense of what's going on and build your own compound conditional statements when you need them.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:
 - `(1 <= 1) and (1 != 1)`
 - `not (1 != 2)`
 - `("good" != "bad") or False`
 - `("good" != "Good") and not (1 == 1)`
2. Add parentheses where necessary so that each of the following expressions evaluates to `True`:
 - `False == not True`
 - `True and False == True and False`
 - `not True and "A" == "B"`

8.3 Control the Flow of Your Program

Now that you can compare values to one another with Boolean comparators and build complex conditional statements with logical operators, you can add some logic to your code so that it performs different actions for different conditions.

The if Statement

An `if` statement tells Python to execute a portion of code only if a certain condition is met.

For example, the following `if` statement will print `2 and 2 is 4` if the conditional `2 + 2 == 4` is `True`:

```
if 2 + 2 == 4:  
    print("2 and 2 is 4")
```

In English, you can read this as “If two plus two is four, then print `2 and 2 is 4`.”

Just like `while` loops, an `if` statement has three parts:

1. The `if` keyword
2. A test condition followed by a colon
3. An indented block of code that is executed if the test condition is `True`

In the above example, the test condition is `2 + 2 == 4`. Since this expression is `True`, executing the `if` statement in IDLE displays the text `2 and 2 is 4`.

If the test condition is `False` (for instance, `2 + 2 == 5`), then Python skips over the indented block of code and continues execution on the next non-indented line.

For example, the following `if` statement doesn't print anything:

```
if 2 + 2 == 5:  
    print("Is this the mirror universe?")
```

A universe where `2 + 2 == 5` is True would be pretty strange indeed!

Note

Leaving off the colon (:) after the test condition in an `if` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 4  
SyntaxError: invalid syntax
```

Once the indented code block in an `if` statement is executed, Python will continue to execute the rest of the program.

Consider the following code:

```
grade = 95  
  
if grade >= 70:  
    print("You passed the class!")  
  
print("Thank you for attending.")
```

The output looks like this:

```
You passed the class!  
Thank you for attending.
```

Since `grade` is 95, the test condition `grade >= 70` is True and the string "You passed the class!" is printed. Then Python executes the rest of the code and prints the string "Thank you for attending."

If you change the value of `grade` to 40, then the output looks like this:

```
Thank you for attending.
```

The line `print("Thank you for attending.")` is executed whether or not `grade` is greater than or equal to 70 because it comes after the indented code block in the `if` statement.

A failing student won't know that they failed if all they see from your code is the text "Thank you for attending." Let's add another `if` statement to tell the student they didn't pass if their grade is less than 70:

```
grade = 40

if grade >= 70:
    print("You passed the class!")

if grade < 70:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

The output now looks like this:

```
You did not pass the class :(
Thank you for attending.
```

In English, we can describe an alternate case with the word *otherwise*. For instance, "If your grade is seventy or above, then you pass the class. Otherwise, you do not pass the class."

Fortunately, there is a keyword that does for Python what the word *otherwise* does in English.

The `else` Keyword

The `else` keyword is used after an `if` statement to execute some code only if the `if` statement's test condition is `False`.

The following code uses `else` to shorten the code in the previous program for displaying whether a student passed a class:

```
grade = 40

if grade >= 70:
    print("You passed the class!")
else:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

In English, the `if` and `else` statements together read as “If the grade is at least seventy, then print You passed the class! Otherwise, print You did not pass the class :(”

Notice that the `else` keyword has no test condition and is followed by a colon. No condition is needed because the `else` statement executes for any condition that fails the `if` statement’s test condition.

Important

Leaving off the colon (:) from the `else` keyword will raise a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... else
SyntaxError: invalid syntax
```

The above example produces the following output:

```
You did not pass the class :(
Thank you for attending.
```

Even if the indented block of code after `else` is executed, Python will still run the line that prints "Thank you for attending."

The `if` and `else` keywords work together nicely if you only need to test a condition with exactly two states. Sometimes, though, you need to check three or more conditions. For that, you use `elif`.

The `elif` Keyword

The `elif` keyword is short for **else if** and can be used to add additional conditions after an `if` statement.

Just like `if` statements, `elif` statements have three parts:

1. The `elif` keyword
2. A test condition followed by a colon
3. An indented code block that is executed if the test condition evaluates to `True`

Important

Leaving off the colon (`:`) at the end of an `elif` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 5:  
...     print("Who broke my math?")  
... elif 2 + 2 == 4  
SyntaxError: invalid syntax
```

The following program combines `if`, `elif`, and `else` to print the letter grade that a student earned in a class:

```
grade = 85 # 1

if grade >= 90: # 2
    print("You passed the class with an A.")
elif grade >= 80: # 3
    print("You passed the class with a B.")
elif grade >= 70: # 4
    print("You passed the class with a C.")
else: # 5
    print("You did not pass the class :(")

print("Thanks for attending.") # 6
```

Both `grade >= 80` and `grade >= 70` are True when `grade` is 85, so you might expect both `elif` blocks on the lines marked # 3 and # 4 to be executed.

However, only the first block for which the test condition is True is executed. All remaining `elif` and `else` blocks are skipped. The program produces the following output:

```
You passed the class with a B.
Thanks for attending.
```

Let's break the code down step by step:

1. `grade` is assigned the value 85 in line 1.
2. `grade >= 90` is False, so the `if` statement in line 2 is skipped.
3. `grade >= 80` is True, so the block under the `elif` statement in line 3 is executed, and "You passed the class with a B." is printed.
4. The `elif` and `else` statements in lines 4 and 5 are skipped since the condition for the `elif` statement on line 3 was met.
5. Finally, line 6 is executed and "Thanks for attending." is printed.

The `if`, `elif`, and `else` keywords are some of the most commonly used keywords in the Python language. They allow you to write code that responds to different conditions with different behavior.

The `if` statement allows you to solve more complex problems than code without conditional logic. You can even nest an `if` statement inside another one to write code that handles tremendously complex logic!

Nested `if` Statements

Just like `for` and `while` loops can be nested within each other, you can nest an `if` statement inside another `if` statement to create complicated decision-making structures.

Consider the following scenario. Two people play a one-on-one sport against each other. You must decide which of the two players wins based on their scores and the sport they're playing:

- If the two players are playing basketball, then the player with the greatest score wins.
- If the two players are playing golf, then the player with the lowest score wins.
- In either sport, if the two scores are equal, then the game is a draw.

The following program solves this problem using nested if statements that reflect the conditions established above:

```
sport = input("Enter a sport: ")
p1_score = int(input("Enter player 1 score: "))
p2_score = int(input("Enter player 2 score: "))

# 1
if sport.lower() == "basketball":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score > p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

# 2
elif sport.lower() == "golf":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score < p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

# 3
else:
    print("Unknown sport")
```

This program first asks the user to input a sport and the scores for two players.

The if statement in #1 executes if sport is equal to "basketball". The .lower() method ensures that user inputs such as "Basketball" or "BasketBall" are interpreted as the same sport.

If both players' scores are equal, then the game is a draw. Otherwise, the player with the highest score wins the game.

The `if` statement in #2 executes if `sport` is equal to "golf". If the two scores are equal, then the game is a draw. Otherwise, the player with the lowest score wins the game.

The `if` statement in #3 executes whenever `sport` is equal to something other than "basketball" or "golf". In that case, the message "Unknown sport" is displayed.

The program's output depends on the input value. Here's a sample execution using "basketball" as the sport:

```
Enter a sport: basketball
Player 1 score: 75
Player 2 score: 64
Player 1 wins.
```

Here's the output with the same player scores and the sport changed to "golf":

```
Enter a sport: golf
Player 1 score: 75
Player 2 score: 64
Player 2 wins.
```

If you enter anything besides `basketball` or `golf` for the sport, then the program displays `Unknown sport`.

Altogether, there are seven possible ways that the program can run:

Sport	Score values
"basketball"	<code>p1_score == p2_score</code>
"basketball"	<code>p1_score > p2_score</code>
"basketball"	<code>p1_score < p2_score</code>
"golf"	<code>p1_score == p2_score</code>
"golf"	<code>p1_score > p2_score</code>
"golf"	<code>p1_score < p2_score</code>
everything else	any combination

Nested `if` statements create many possible **execution paths** for your code. If you have more than a couple of levels of nested `if` statements, then the number of possible ways the code can execute grows quickly.

Note

The complexity that results from using deeply nested `if` statements may make it difficult to predict how your program will behave under given conditions.

For this reason, nested `if` statements are generally discouraged.

Let's see how to simplify the previous program by removing nested `if` statements.

First, regardless of the sport, the game is a draw if `p1_score` is equal to `p2_score`. So, we can move the check for equality out from the nested `if` statements under each sport to make a single `if` statement:

```
if p1_score == p2_score:  
    print("The game is a draw.")  
  
elif sport.lower() == "basketball":  
    if p1_score > p2_score:  
        print("Player 1 wins.")  
    else:  
        print("Player 2 wins.")  
  
elif sport.lower() == "golf":  
    if p1_score < p2_score:  
        print("Player 1 wins.")  
    else:  
        print("Player 2 wins.")  
  
else:  
    print("Unknown sport.")
```

Now there are only six ways that the program can execute.

That's still quite a few ways. Can you think of another way to make the program simpler?

Here's one way to simplify it. Player 1 wins if either of the following conditions is met:

1. sport is "basketball" and p1_score is greater than p2_score.
2. sport is "golf" and p1_score is less than p2_score.

We can describe this with compound conditional expressions:

```
sport = sport.lower()
p1_wins_bball = (sport == "basketball") and (p1_score > p2_score)
p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
p1_wins = player1_wins_basketball or player1_wins_golf
```

This code is dense, so let's walk through it one step at a time.

First, the string assigned to `sport` is converted to all lowercase so that we can compare the value to other strings without worrying about errors due to case.

On the next line, we have a structure that might look a little strange. There is an assignment operator (`=`) followed by an expression with the equality comparator (`==`). This line evaluates the following compound logical expression and assigns its value to the `p1_wins_bball` variable:

```
(sport == "basketball") and (p1_score > p2_score)
```

If `sport` is "basketball" and `p1_score` score is greater than `p2_score`, then `p1_wins_bball` is True.

Next, a similar operation is done for the `p1_wins_golf` variable. If `sport` is "golf" and `p1_score` is less than `p2_score`, then `p1_wins_golf` is True.

Finally, `p1_wins` will be True if player 1 wins the basketball or golf game and will be False otherwise.

Using this code, you can simplify the program quite a bit:

```
sport = sport.lower()

if p1_score == p2_score:
    print("The game is a draw.")

elif (sport == "basketball") or (sport == "golf"):
    p1_wins_bball = (sport == "basketball") and (p1_score > p2_score)
    p1_wins_golf = (sport == "golf") and (p1_score < p2_score)
    p1_wins = p1_wins_bball or p1_wins_golf

    if p1_wins:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

else:
    print("Unknown sport")
```

In this revised version of the program, there are only four ways the program can execute, and the code is easier to understand.

Nested `if` statements are sometimes necessary. However, if you find yourself writing lots of nested `if` statements, then it might be a good idea to stop and think about how you might simplify your code.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that prompts the user to enter a word using the `input()` function and compares the length of the word to the number five. The program should display one of the following outputs, depending on the length of the user's input:
 - "Your input is less than 5 characters long"
 - "Your input is greater than 5 characters long"
 - "Your input is 5 characters long"

2. Write a program that displays "I'm thinking of a number between 1 and 10. Guess which one." Then use `input()` to get a number from the user. If the user inputs the number 3, then the program should display "You win!" For any other input, the program should display "You lose."

8.4 Challenge: Find the Factors of a Number

A factor of a positive integer n is any positive integer less than or equal to n that divides n with no remainder.

For example, 3 is a factor of 12 because 12 divided by 3 is 4 with no remainder. However, 5 is not a factor of 12 because 5 goes into 12 twice with a remainder of 2.

Write a program called `factors.py` that asks the user to input a positive integer and then prints out the factors of that number. Here's a sample run of the program with output:

```
Enter a positive integer: 12
1 is a factor of 12
2 is a factor of 12
3 is a factor of 12
4 is a factor of 12
6 is a factor of 12
12 is a factor of 12
```

Hint: Recall from chapter 5 that you can use the `%` operator to get the remainder of dividing one number by another.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

8.5 Break Out of the Pattern

In chapter 6, you learned how to repeat a block of code many times using a `for` or `while` loop. Loops are useful for performing a repetitive task and for processing many different inputs in the same way.

In this section, you'll learn how to write `if` statements that are nested in `for` loops. You'll also learn about two keywords—`break` and `continue`—that allow you to precisely control the flow of execution through a loop.

if Statements and for Loops

The block of code in a `for` loop is just like any other block of code. That means you can nest an `if` statement in a `for` loop just like you can anywhere else in your code.

The following example uses a `for` loop with an `if` statement to compute and display the sum of all even integers less than one hundred:

```
sum_of_evens = 0

for n in range(101):
    if n % 2 == 0:
        sum_of_evens = sum_of_evens + n

print(sum_of_evens)
```

First, `sum_of_evens` is initialized to 0. Then the program loops over the numbers 0 to 100 and adds the even values to `sum_of_evens`. The final value of `sum_of_evens` is 2550.

break

The `break` keyword tells Python to literally break out of a loop. That is, the loop stops completely and any code that comes after the loop is executed.

For example, the following code loops over the numbers 0 to 3, but stops the loop when the number 2 is encountered:

```
for n in range(4):
    if n == 2:
        break
    print(n)

print(f"Finished with n = {n}")
```

Only the first two numbers are printed in the output:

```
0
1
Finished with n = 2
```

continue

The `continue` keyword is used to skip any remaining code in the loop body and continue on to the next iteration.

For example, the following code loops over the numbers 0 to 3, printing each number as is goes but skipping the number 2:

```
for i in range(4):
    if i == 2:
        continue
    print(i)

print(f"Finished with i = {i}")
```

All the numbers except for 2 are printed in the output:

```
0
1
3
Finished with i = 3
```

Note

It's always a good idea to give short but descriptive names to your variables that make it easy to tell what they're supposed to represent.

The letters `i`, `j`, `k`, and `n` are exceptions because they're so common in programming.

These letters are almost always used when we need a throwaway variable solely for the purpose of keeping count while working through a loop.

To summarize, the `break` keyword is used to stop a loop if a certain condition is met, and the `continue` keyword is used to skip any remaining code in a loop if a certain condition is met.

for ... else Loops

Although this structure isn't used very frequently, loops can have their own `else` clause in Python. Let's look at an example:

```
phrase = "it marks the spot"

for character in phrase:
    if character == "X":
        break
else:
    print("There was no 'X' in the phrase")
```

The `for` loop in this example loops over the characters in the phrase "it marks the spot" and stops if the letter "X" is found. If you run the code in the example, then you'll see that `There was no 'X' in the phrase` is printed to the console.

Now change `phrase` to the string "X marks the spot". When you run the same code with this phrase, there's no output. What's going on?

Any code in the `else` block is executed only if the preceding `for` loop completes without encountering a `break` statement.

So, when you run the code with `phrase = "it marks the spot"`, the line of code containing the `break` statement is never run since there is no `x` character in the phrase. Instead, the `else` block is executed and the string "There was no 'X' in the phrase" is displayed.

On the other hand, when you run the code with `phrase = "X marks the spot"`, the line containing the `break` statement *does* get executed, so the `else` block is never run, and no output gets displayed.

Here's a practical example that gives a user three attempts to enter a password:

```
for n in range(3):
    password = input("Password: ")
    if password == "I<3Bieber":
        break
    print("Password is incorrect.")
else:
    print("Suspicious activity. The authorities have been alerted.")
```

This example loops over the numbers 0 to 2. On each iteration, the user is prompted to enter a password. If the password entered is correct, then `break` is used to exit the loop. Otherwise, the user is told that the password is incorrect and is given another attempt.

After three unsuccessful attempts, the `for` loop terminates without ever executing the line of code containing `break`. In that case, the `else` block is executed and the user is warned that the authorities have been alerted.

Note

We focused on `for` loops in this section because they're generally the most common kind of loops.

However, everything discussed here also works for `while` loops. That is, you can use `break` and `continue` inside a `while` loop. `while` loops can even have an `else` clause!

Using conditional logic inside the body of a loop opens up several possibilities for controlling how your code executes. You can stop loops with the `break` keyword or skip an iteration with `continue`. You can even make sure some code runs only if a loop completes without ever encountering a `break` statement.

These are some powerful tools to have in your tool kit!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Using `break`, write a program that repeatedly asks the user for some input and quits only if the user enters "q" or "Q".
2. Using `continue`, write a program that loops over the numbers 1 to 50 and prints all numbers that are not multiples of 3.

8.6 Recover From Errors

Encountering errors in your code might be frustrating, but it's totally normal! It happens to even the best programmers.

Programmers often refer to runtime errors as **exceptions**. So, when you encounter an error, congratulate yourself! You've just made the code do something exceptional.

To create robust programs, you need to be able to handle errors caused

by invalid user input or any other unpredictable source. In this section, you'll learn how to do just that.

A Zoo of Exceptions

When you encounter an exception, it's useful to know what went wrong. Python has a number of built-in exception types that describe different kinds of errors.

You've seen several different errors throughout this book. Let's collect them here and add a few new ones to the list.

ValueError

A `ValueError` occurs when an operation encounters an invalid value. For example, trying to convert the string "not a number" to an integer results in a `ValueError`:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("not a number")
ValueError: invalid literal for int() with base 10: 'not a number'
```

The last line displays the name of the exception and a description of the problem. This is the general format for all Python exceptions.

TypeError

A `TypeError` occurs when an operation is performed on a value of the wrong type. For example, trying to add a string and an integer will result in a `TypeError`:

```
>>> "1" + 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    "1" + 2
TypeError: can only concatenate str (not "int") to str
```

NameError

A `NameError` occurs when you try to use a variable name that hasn't been defined:

```
>>> print(does_not_exist)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(does_not_exist)
NameError: name 'does_not_exist' is not defined
```

ZeroDivisionError

A `ZeroDivisionError` occurs when the divisor in a division operation is zero:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

OverflowError

An `OverflowError` occurs when the result of an arithmetic operation is too large. For example, if you tried to raise the value 2.0 to the power 1_000_000, then you'd get an `OverflowError`:

```
>>> pow(2.0, 1_000_000)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    pow(2.0, 1_000_000)
OverflowError: (34, 'Result too large')
```

You may recall from chapter 5 that integers in Python have unlimited precision. This means that `OverflowErrors` can occur only with floating-point numbers. Raising the *integer* 2 to the value 1_000_000 will not raise an `OverflowError`!

A full list of Python's built-in exceptions can be found [in the official docs](#).

The try and except Keywords

Sometimes you can predict that a certain exception might occur. Instead of letting the program crash, you can catch the error when it occurs and do something else instead.

For example, you might need to ask the user to input an integer. If the user enters a non-integer value, such as the string "a", then you need to let them know that they entered an invalid value.

To prevent the program from crashing, you can use the `try` and `except` keywords. Let's look at an example:

```
try:  
    number = int(input("Enter an integer: "))  
except ValueError:  
    print("That was not an integer")
```

The `try` keyword indicates a `try` block and is followed by a colon. The code indented after `try` is executed. In this case, the user is asked to input an integer. Since `input()` returns a string, the user input is converted to an integer with `int()`, and the result is assigned to the variable `number`.

If the user inputs a non-integer value, then the `int()` operation will raise a `ValueError`. If that happens, then the code indented below the line `except ValueError` will be executed. So, instead of the program crashing, the string "That was not an integer" will be displayed.

If the user does input a valid integer value, then the code in the `except ValueError` block will never be executed.

However, if a *different* type of exception occurs, such as a `TypeError`, then the program will crash. The above example handles only one type of exception—a `ValueError`.

You can handle multiple exception types by separating the exception names with commas and putting the list of names in parentheses:

```
def divide(num1, num2):
    try:
        print(num1 / num2)
    except (TypeError, ZeroDivisionError):
        print("encountered a problem")
```

In this example, `divide()` takes two parameters, `num1` and `num2`, and prints the result of dividing `num1` by `num2`.

If `divide()` is called with an argument that is a string, then the division operation will raise a `TypeError`. Additionally, if `num2` is zero, then a `ZeroDivisionError` will be raised.

The line `except (TypeError, ZeroDivisionError)` will handle both of these exceptions and display the string "encountered a problem" if either exception is raised.

Many times, though, it's helpful to catch each error individually so that you can display text that is more helpful to the user. To do this, you can use multiple `except` blocks after a `try` block:

```
def divide(num1, num2):
    try:
        print(num1 / num2)
    except TypeError:
        print("Both arguments must be numbers")
    except ZeroDivisionError:
        print("num2 must not be 0")
```

In this example, the `TypeError` and `ZeroDivisionError` are handled separately. This way, a more descriptive message is displayed if something goes wrong.

If one of `num1` or `num2` is not a number, then a `TypeError` is raised and the message "Both arguments must be numbers" is displayed. If `num2` is zero,

then a `ZeroDivisionError` is raised and the message "num2 must not be 0" is displayed.

The Bare `except` Clause

You can use the `except` keyword by itself without naming specific exceptions:

```
try:  
    # Do lots of hazardous things that might break  
except:  
    print("Something bad happened!")
```

If any exception is raised while executing the code in the `try` block, then the `except` block will run and the message "Something bad happened!" will be displayed.

This might sound like a great way to ensure your program never crashes, **but this is actually bad idea and the pattern is generally frowned upon!**

There are a couple of reasons for this, but the most important reason for new programmers is that catching every exception could hide bugs in your code, giving you a false sense of confidence that your code works as expected.

If you catch only specific exceptions, then when unexpected errors are encountered, Python will print the traceback and error details, giving you more information to work with when debugging your code.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that repeatedly asks the user to input an integer. If the user enters something other than an integer, then the program should catch the `ValueError` and display the message "Try again."

Once the user enters an integer, the program should display the number back to the user and end without crashing.

2. Write a program that asks the user to input a string and an integer n , then displays the character at index n in the string.

Use error handling to make sure the program doesn't crash if the user enters something other than an integer or if the index is out of bounds. The program should display a different message depending on which error occurs.

8.7 Simulate Events and Calculate Probabilities

In this section, you'll apply some of the concepts you've learned about loops and conditional logic to a real-world problem: simulating events and calculating probabilities.

You'll be running a simple simulation known as a **Monte Carlo experiment**. Each experiment consists of a **trial**, which is just some process that can be repeated, such as flipping a coin. Each trial generates an **outcome**, such as landing on heads or tails. The trial is repeated over and over again in order to calculate the probability that a given outcome occurs.

In order to do this, you'll need to add some randomness to your code.

The `random` module

Python provides several functions for generating random numbers in the `random` module. A **module** is a collection of related code. Python's **standard library** is an organized collection of modules that you can **import** into your own code to solve various problems.

To import the `random` module, type the following into IDLE's interactive window:

```
>>> import random
```

Now you can use functions from the `random` module in your code.

Note

You'll learn more about modules and `import` statements in chapter 11, "Modules and Packages."

The `randint()` function in the `random` module has two required parameters called `a` and `b`, both of which must be integers. The function returns a random integer that is greater than or equal to `a` and less than or equal to `b`.

For example, the following code produces a random integer between 1 and 10:

```
>>> random.randint(1, 10)
9
```

Since the result is random, your output will probably be different than 9. If you type the same code in again, you will likely get a different number.

Since `randint()` is located in the `random` module, you must type `random` followed by a dot (.) and then the function name in order to use it.

When using `randint()`, it's important to remember that the two parameters `a` and `b` must both be integers, and the output might be equal to `a`, `b`, or any number in between. For instance, `random.randint(0, 1)` randomly returns either 0 or 1.

Furthermore, each integer between `a` and `b` is equally likely to be returned by `randint()`. So, for `randint(1, 10)`, each integer between 1 and 10 has a 10 percent chance of being returned. For `randint(0, 1)`, there is a 50 percent chance that 0 will be returned.

Fair Coins

Let's see how to use `randint()` to simulate flipping a fair coin. By *fair coin* we mean a coin that, when flipped, has an equal chance of landing on heads or tails.

One trial for your experiment will be a single coin flip. The outcome will be either a heads or a tails. The question is, in general, over many coin flips, what is the ratio of heads to tails?

Let's think about how to solve this problem. You'll need to keep track of how many times you get a heads and tails, so you need a heads tally and a tails tally. Each trial has two steps:

1. Flip the coin.
2. If the coin lands on heads, then update the heads tally. Otherwise, the coin landed on tails, so update the tails tally.

You need to repeat the trial many times, say ten thousand. A `for` loop over `range(10_000)` is a good choice for doing something like that.

Now that you have a plan, you can start by writing a function called `coin_flip()` that randomly returns the string "heads" or the string "tails". You can do this using `random.randint(0, 1)`. You can use 0 for heads and 1 for tails.

Here's the code for the `coin_flip()` function:

```
import random

def coin_flip():
    """Randomly return 'heads' or 'tails'."""
    if random.randint(0, 1) == 0:
        return "heads"
    else:
        return "tails"
```

If `random.randint(0, 1)` returns a 0, then `coin_flip()` returns "heads". Otherwise, `coin_flip()` returns "tails".

Now you can write a `for` loop that flips the coin ten thousand times and updates the heads or tails tally accordingly:

```
# First initialize the tallies to 0
heads_tally = 0
tails_tally = 0

for trial in range(10_000):
    if coin_flip() == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1
```

First, the two variables `heads_tally` and `tails_tally` are created and both are initialized to the integer 0.

Then the `for` loop runs ten thousand times, calling `coin_flip()` each time. If `coin_flip()` returns the string "heads", then the `heads_tally` variable is incremented by 1. Otherwise `tails_tally` is incremented by 1.

Finally, you can print the ratio of heads and tails:

```
ratio = heads_tally / tails_tally
print(f"The ratio of heads to tails is {ratio}")
```

If you save the above code to a file and run it a few times, then you'll see that the result is usually between .98 and 1.02. If you increase the `range(10_000)` in the `for` loop to, say, `range(50_000)`, then the results should get closer to 1.0.

This behavior makes sense. Since the coin is fair, you should expect that after many flips, the number of heads will be roughly equal to the number of tails.

In life, though, things aren't always fair. A coin may have a slight tendency to land on heads instead of tails, or vice versa. So, how do you simulate something like an unfair coin?

Unfair Coins

`randint()` returns a 0 or a 1 with equal probability. If 0 represents tails and 1 represents heads, then to simulate an unfair coin, you need a way to increase the probability of returning either 0 or 1.

The `random()` function takes no arguments and returns a floating-point number greater than or equal to 0.0 but less than 1.0. Each possible return value is equally likely. In probability theory, this is known as a **uniform probability distribution**.

One consequence of this is that, given a number n between 0 and 1, the probability that `random()` returns a number less than n is just n itself. For example, the probability that `random()` is less than .8 is .8, and the probability that `random()` is less than .25 is .25.

Using this fact, you can write a function that simulates a coin flip but returns tails with a specified probability:

```
import random

def unfair_coin_flip(probability_of_tails):
    if random.random() < probability_of_tails:
        return "tails"
    else:
        return "heads"
```

For example, `unfair_coin_flip(.7)` has a 70 percent chance of returning "tails".

Let's rewrite the coin flip experiment that you created earlier, using `unfair_coin_flip()` to run each trial with an unfair coin:

```
heads_tally = 0
tails_tally= 0

for trial in range(10_000):
    if unfair_coin_flip(.7) == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1

ratio = heads_tally / tails_tally
print(f"The ratio of heads to tails is {ratio}")
```

Running this simulation a few times shows that the ratio of heads to tails has gone down from 1 in the experiment with a fair coin to about .43.

In this section, you learned about the `randint()` and `random()` functions in the `random` module and saw how to use conditional logic and loops to write some coin toss simulations. Simulations like these are used in numerous disciplines to make predictions and to test computer models of real-world events.

The `random` module provides many useful functions for generating random numbers and writing simulations. You can learn more about `random` in *Real Python*'s “[Generating Random Data in Python \(Guide\)](#).”

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a function called `roll()` that uses `randint()` to simulate rolling a fair die by returning a random integer between 1 and 6.
2. Write a program that simulates ten thousand rolls of a fair die and displays the average number rolled.

8.8 Challenge: Simulate a Coin Toss Experiment

Suppose you flip a fair coin repeatedly until it lands on heads and tails at least one time each. In other words, after the first flip, you continue to flip the coin until it lands on the other side.

Doing this generates a sequence of heads and tails. For example, the first time you do this experiment, the sequence might be heads, heads, tails.

On average, how many flips are needed for the sequence to contain both heads and tails?

Write a simulation that runs ten thousand trials of the experiment and prints the average number of flips per trial.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

8.9 Challenge: Simulate an Election

With some help from the `random` module and a little conditional logic, you can simulate an election between two candidates.

Suppose two candidates, Candidate A and Candidate B, are running for mayor in a city with three voting regions. The most recent polls show that Candidate A has the following chances for winning in each region:

- **Region 1:** 87 percent chance of winning
- **Region 2:** 65 percent chance of winning
- **Region 3:** 17 percent chance of winning

Write a program that simulates the election ten thousand times and prints the percentage of times in which Candidate A wins.

To keep things simple, assume that a candidate wins the election if they win in at least two of the three regions.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

8.10 Summary and Additional Resources

In this chapter, you learned about conditional statements and conditional logic. You saw how to compare values using comparison operators like `<`, `>`, `<=`, `>=`, `!=`, and `==`. You also saw how to build complex conditional statements using `and`, `or`, and `not`.

Next, you saw how to control the flow of your program using `if` statements. You learned how to create branches in your program using `if ... else` and `if ... elif ... else`. You also learned how to control precisely how code is executed inside an `if` block using `break` and `continue`.

You learned about the `try ... except` pattern to handle errors that may occur during runtime. This is an important construct that allows your programs to handle the unexpected gracefully and keep users happy that the program didn't crash.

Finally, you applied the techniques that you learned in this chapter and used the `random` module to build some simple simulations.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-conditional-logic

Additional Resources

A wise Vulcan once said:

Logic is the beginning of wisdom . . . not the end.

— Spock, *Star Trek*

Check out the following resources to learn more about conditional logic:

- “[Operators and Expressions in Python](#)”
- “[Conditional Statements in Python](#)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 9

Tuples, Lists, and Dictionaries

So far, you've been working with fundamental data types like `str`, `int`, and `float`. Many real-world problems are easier to solve when you combine simple data types into more complex data structures.

A **data structure** models a collection of data, such as a list of numbers, a row in a spreadsheet, or a record in a database. Modeling the data that your program interacts with using the right data structure is often the key to writing simple and effective code.

Python has three built-in data structures that are the focus of this chapter: tuples, lists, and dictionaries.

In this chapter, you'll learn:

- How to work with tuples, lists, and dictionaries
- What immutability is and why it's important
- When to use different data structures

Let's dive in!

9.1 Tuples Are Immutable Sequences

Perhaps the simplest compound data structure is a sequence of items.

A **sequence** is an ordered list of values. Each element in a sequence is assigned an **index**, which is an integer that indicates the element's position in the sequence. Just like strings, the index of the first value in a sequence is 0.

For example, the letters of the English alphabet form a sequence whose first element is *A* and last element is *Z*. Strings are also sequences. The string "Python" has six elements, starting with "P" at index 0 and ending with "n" at index 5.

Some real-world examples of sequences include the sequence of values measured by a sensor every second, the sequence of a student's test scores throughout the school year, or the sequence of daily stock prices for some company over a period of time.

In this section, you'll learn how to use Python's built-in `tuple` data type to create sequences of values.

What Is a Tuple?

The word **tuple** comes from mathematics, where it's used to describe a finite ordered sequence of values.

Usually, mathematicians write tuples by listing each element, separated by a comma, inside a pair of parentheses. For example, `(1, 2, 3)` is a tuple containing three integers.

Tuples are **ordered** because their elements appear in an ordered fashion. The first element of `(1, 2, 3)` is 1, the second element is 2, and the third is 3.

Python borrows both the name and the notation for tuples from mathematics.

How to Create a Tuple

There are a few ways to create a tuple in Python. We'll cover two of them:

1. Tuple literals
2. The built-in `tuple()`

Tuple Literals

Just like a string literal is a string that is explicitly created by surrounding some text with quotes, a **tuple literal** is a tuple that is written out explicitly as a comma-separated list of values surrounded by parentheses.

Here's an example of a tuple literal:

```
>>> my_first_tuple = (1, 2, 3)
```

This creates a tuple containing the integers 1, 2, and 3 and assigns it to the name `my_first_tuple`.

You can check that `my_first_tuple` is a tuple by using `type()`:

```
>>> type(my_first_tuple)
<class 'tuple'>
```

Unlike strings, which are sequences of characters, tuples may contain any type of value, including values of different types. The tuple `(1, 2.0, "three")` is perfectly valid.

There's also a special tuple that doesn't contain any values. This tuple is called the **empty tuple** and can be created by typing two parentheses with nothing between them:

```
>>> empty_tuple = ()
```

At first glance, the empty tuple may seem like a strange and useless concept, but it's actually quite practical.

For example, suppose you’re asked to provide a tuple containing all integers that are simultaneously even *and* odd. No such integer exists, but the empty tuple allows you to provide the requested tuple.

How do you think you would create a tuple with exactly one element? Try out the following in IDLE:

```
>>> x = (1)
>>> type(x)
<class 'int'>
```

When you surround a value with parentheses but don’t include any commas, Python interprets the value not as a tuple but as the type of value inside the parentheses. So, in this case, (1) is just a weird way of writing the integer 1.

To create a tuple containing the single value 1, you need to include a comma after the 1:

```
>>> x = (1,)
>>> type(x)
<class 'tuple'>
```

A tuple containing a single element might seem as strange as the empty tuple. Can’t you drop all this tuple business and just use the value itself?

It all depends on the problem you’re solving.

If you’re asked to provide a tuple of all even prime numbers, then you must provide the tuple (2,) since 2 is the only even prime number. The value 2 isn’t a good solution because it isn’t a tuple.

This might seem overly pedantic, but programming often involves a certain amount of pedantry. Computers are, after all, the ultimate pedants.

The Built-In tuple()

You can also use the built-in `tuple()` to create a tuple from another sequence type, such as a string:

```
>>> tuple("Python")
('P', 'y', 't', 'h', 'o', 'n')
```

`tuple()` only accepts a single parameter, so you can't just list the values you want in the tuple as individual arguments. If you do, then Python raises a `TypeError`:

```
>>> tuple(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    tuple(1, 2, 3)
TypeError: tuple expected at most 1 arguments, got 3
```

You'll also get a `TypeError` if the argument passed to `tuple()` can't be interpreted as a list of values:

```
>>> tuple(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    tuple(1)
TypeError: 'int' object is not iterable
```

The word **iterable** in the error message indicates that a single integer can't be **iterated**, which is to say that the integer data type doesn't contain multiple values that can be accessed one by one.

The single parameter of `tuple()` is optional, though, and leaving it out produces an empty tuple:

```
>>> tuple()
()
```

However, most Python programmers prefer to use the shorter `()` for creating an empty tuple.

Similarities Between Tuples and Strings

Tuples and strings have a lot in common. Both are sequence types with finite lengths, both support indexing and slicing, both are immutable, and both can be iterated over in a loop.

The main difference between strings and tuples is that the elements of tuples can be any kind of value you like, whereas strings can only contain characters.

Let's look at some of the parallels between strings and tuples in more depth.

Tuples Have a Length

Both strings and tuples have a **length**. The length of a string is the number of characters in it. The length of a tuple is the number of elements it contains.

Just like with strings, you can use `len()` to determine the length of a tuple:

```
>>> numbers = (1, 2, 3)
>>> len(numbers)
3
```

Tuples Support Indexing and Slicing

Recall from chapter 4 that you can access a character in a string using index notation:

```
>>> name = "David"
>>> name[1]
'a'
```

The index notation `[1]` after the variable `name` tells Python to get the character at index 1 in the string "David". Since counting starts at zero, the character at index 1 is the letter "a".

Tuples also support index notation:

```
>>> values = (1, 3, 5, 7, 9)  
>>> values[2]  
5
```

Another feature that strings and tuples have in common is slicing. Recall that you can extract a substring from a string using slice notation:

```
>>> name = "David"  
>>> name[2:4]  
"vi"
```

The slice notation [2:4] after the variable `name` creates a new string containing the characters in `name` starting at position 2 and going up to but not including the character at position 4.

Slice notation also works with tuples:

```
>>> values = (1, 3, 5, 7, 9)  
>>> values[2:4]  
(5, 7)
```

The slice `values[2:4]` creates a new tuple containing all the integers in `values` starting from position 2 and going up to but not including position 4.

The same rules governing string slices also apply to tuple slices. You may want to take some time to review the slicing examples in chapter 4 with some of your own examples of tuples.

Tuples Are Immutable

Like strings, tuples are immutable. This means you can't change the value of an element in a tuple once it's been created.

If you do try to change the value at some index of a tuple, then Python will raise a `TypeError`:

```
>>> values[0] = 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    values[0] = 2
TypeError: 'tuple' object does not support item assignment
```

Note

Although tuples are immutable, there are some situations in which the values in a tuple can change.

These quirks and oddities are covered in depth in *Real Python's Immutability in Python* video course.

Tuples Are Iterable

Just like strings, tuples are **iterable**, so you can loop over them:

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> for vowel in vowels:
...     print(vowel.upper())
...
A
E
I
O
U
```

The `for` loop in this example works just like the `for` loops you saw in chapter 6 that loop over a `range()` of numbers.

On the first step of the loop, the value "a" is extracted from the tuple `vowels`. It's converted to an uppercase letter using the `.upper()` string method you learned about in chapter 4 and then displayed with `print()`.

The next step of the loop extracts the value "e", converts it to uppercase, and prints it. This continues for each of the values "i", "o", and "u".

Now that you've seen how to create tuples and some of the basic operations they support, let's look at some common use cases.

Tuple Unpacking

There's a third, although less common, way of creating a tuple. You can create a literal tuple without the parentheses:

```
>>> coordinates = 4.21, 9.29
>>> type(coordinates)
<class 'tuple'>
>>> coordinates
(4.21, 9.29)
```

You can verify that `coordinates` is indeed a tuple with `type()`, and confirm that Python represents it the same way as a tuple that you create with the surrounding parentheses.

Independent of how you created your tuple, you can **unpack** the values in the tuple:

```
>>> x, y = coordinates
>>> x
4.21
>>> y
9.29
```

Here, you **unpacked** the values contained in the single tuple `coordinates` into two distinct variables, `x` and `y`.

With tuple unpacking, you can make multiple variable assignments in a single line:

```
>>> name, age, occupation = "Adisa", 34, "programmer"
>>> name
'Adisa'
>>> age
34
>>> occupation
'programmer'
```

This works because first, on the right side of the assignment, you create a literal tuple with the values "Adisa", 34, and "programmer". Then, you unpack the values in that tuple into the three variables `name`, `age`, and `occupation`, in that order.

Note

While assigning multiple variables in a single line can shorten the number of lines in a program, you may want to refrain from assigning too many values in a single line.

Assigning more than two or three variables this way can make it difficult to tell which value is assigned to which variable name.

Keep in mind that the number of variable names on the left side of the assignment statement must equal the number of values in the tuple on the right side. Otherwise, Python raises a `ValueError`:

```
>>> a, b, c, d = 1, 2, 3
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a, b, c, d = 1, 2, 3
ValueError: not enough values to unpack (expected 4, got 3)
```

The error message here tells you that the tuple on the right side doesn't have enough values to unpack into the four variable names on the left.

Python also raises a `ValueError` if the number of values in the tuple exceeds the number of variable names:

```
>>> a, b, c = 1, 2, 3, 4
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    a, b, c = 1, 2, 3, 4
ValueError: too many values to unpack (expected 3)
```

Now the error message indicates that there are too many values in the tuple to unpack into three variables.

Checking the Existence of Values With `in`

You can check whether a value is contained in a tuple with the `in` keyword:

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> "o" in vowels
True
>>> "x" in vowels
False
```

If the value to the left of `in` is contained in the tuple to the right, then the result is `True`. Otherwise, the result is `False`.

Returning Multiple Values From a Function

One common use of tuples is to return multiple values from a single function:

```
>>> def adder_subtractor(num1, num2):
...     return (num1 + num2, num1 - num2)
...
>>> adder_subtractor(3, 2)
(5, 1)
```

The function `adder_subtractor()` has two parameters, `num1` and `num2`. It returns a tuple in which the first element is the sum of the two num-

bers and the second element is the difference between the two numbers.

Strings and tuples are just two of Python’s built-in sequence types. Both are immutable and iterable, and both can be used with either index or slice notation.

In the next section, you’ll learn about a third sequence type with one very big difference from strings and tuples: mutability.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a tuple literal named `cardinal_numbers` that holds the strings "first", "second", and "third", in that order.
2. Using index notation and `print()`, display the string at index 1 in `cardinal_numbers`.
3. In a single line of code, unpack the values in `cardinal_numbers` into three new strings named `position1`, `position2`, and `position3`. Then print each value on a separate line.
4. Using `tuple()` and a string literal, create a tuple called `my_name` that contains the letters of your name.
5. Check whether the character "x" is in `my_name` using the `in` keyword.
6. Create a new tuple containing all but the first letter in `my_name` using slice notation.

9.2 Lists Are Mutable Sequences

The `list` data structure is another sequence type in Python. Just like strings and tuples, lists contain items that are indexed by integers, starting with 0.

On the surface, lists look and behave a lot like tuples. With lists, you can use index and slice notation, check for the existence of an element using `in`, and iterate over elements using a `for` loop.

Unlike tuples, however, lists are **mutable**, meaning you can change the value at a given index even after the list has been created.

In this section, you'll learn how to create lists and compare them with tuples.

Creating Lists

A **list literal** looks almost exactly like a tuple literal, except that it's surrounded by square brackets (`[]`) instead of parentheses:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> type(colors)
<class 'list'>
```

When you inspect a list, Python displays it as a list literal:

```
>>> colors
['red', 'yellow', 'green', 'blue']
```

Like tuple values, list values are not required to be of the same type. The list literal `["one", 2, 3.0]` is perfectly valid.

Aside from list literals, you can also use the built-in `list()` to create a new list object from any other sequence. For instance, the tuple `(1, 2, 3)` can be passed to `list()` to create the list `[1, 2, 3]`:

```
>>> list((1, 2, 3))
[1, 2, 3]
```

You can even create a list from a string:

```
>>> list("Python")
['P', 'y', 't', 'h', 'o', 'n']
```

Each letter in the string becomes an element of the list.

There's a more useful way to create a list from a string. You can create a list from a string of comma-separated list items using the string object's `.split()` method:

```
>>> groceries = "eggs, milk, cheese"
>>> grocery_list = groceries.split(", ")
>>> grocery_list
['eggs', 'milk', 'cheese']
```

The string argument passed to `.split()` is called the **separator**. By changing the separator, you can split strings into lists in numerous ways:

```
>>> # Split string on semicolons
>>> "a;b;c".split(";")
['a', 'b', 'c']

>>> # Split string on spaces
>>> "The quick brown fox".split(" ")
['The', 'quick', 'brown', 'fox']

>>> # Split string on multiple characters
>>> "abbaabba".split("ba")
['ab', 'ab', '']
```

In the last example above, the string is split around occurrences of the substring "ba", which occurs first at index 2 and again at index 6. Since the separator has two characters, only the characters at indices 0, 1, 4, and 5 become elements of the list.

`.split()` always returns a list whose length is one more than the number of separators contained in the string. The separator "ba" appears twice in "abbaabba", so the list returned by `split()` has three elements.

Notice that the last element of the list is the empty string. That happens because the final "ba" isn't followed by any other characters.

If the separator isn't contained in the string at all, then `.split()` returns a list with the string as its only element:

```
>>> "abbaabba".split("c")
['abbaabba']
```

In all, you've seen three ways to create a list:

1. A list literal
2. The built-in `list()`
3. The string `.split()` method

Lists support the same operations supported by tuples.

Basic List Operations

Indexing and slicing operations work on lists the same way they do on tuples.

You can access list elements using index notation:

```
>>> numbers = [1, 2, 3, 4]
>>> numbers[1]
2
```

You can create a new list from an existing one using slice notation:

```
>>> numbers[1:3]
[2, 3]
```

You can check for the existence of list elements using the `in` operator:

```
>>> # Check existence of an element
>>> "Bob" in numbers
False
```

Lists are iterable, which means you can iterate over them with a `for` loop:

```
>>> # Print only the even numbers in the list
>>> for number in numbers:
...     if number % 2 == 0:
...         print(number)
...
2
4
```

The major difference between lists and tuples is that list elements can be changed, but tuple elements cannot.

Changing Elements in a List

You can think of a list as a sequence of numbered slots. Each slot holds a value, and every slot must be filled at all times. But you can swap out the value in a given slot with a new one whenever you want.

The ability to swap values in a list for other values is called **mutability**. Lists are **mutable**. The elements of tuples can't be swapped for new values, so tuples are said to be **immutable**.

To swap one value in a list with another, assign the new value to a slot using index notation:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[0] = "burgundy"
```

The value at index 0 changes from "red" to "burgundy":

```
>>> colors
['burgundy', 'yellow', 'green', 'blue']
```

You can change several values in a list at once with a **slice assignment**:

```
>>> colors[1:3] = ["orange", "magenta"]
>>> colors
['burgundy', 'orange', 'magenta', 'blue']
```

`colors[1:3]` selects the slots with indices 1 and 2. The values in these slots are assigned to "orange" and "magenta", respectively.

The list assigned to a slice doesn't need to have the same length as the slice. For instance, you can assign a list of three elements to a slice with two elements:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[1:3] = ["orange", "magenta", "aqua"]
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
```

The values "orange" and "magenta" replace the original values "yellow" and "green" in `colors` at the indices 1 and 2. Then a new slot is created at index 4 and "blue" is assigned to this index. Finally, "aqua" is assigned to index 3.

If the length of the list assigned to the slice is less than the length of the slice, then the overall length of the original list will be reduced:

```
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
>>> colors[1:4] = ["yellow", "green"]
>>> colors
['red', 'yellow', 'green', 'blue']
```

The values "yellow" and "green" replace the values "orange" and "magenta" in `colors` at the indices 1 and 2. Then the value "blue" fills in at index 3, and index 4 is removed from `colors` entirely.

The above examples show how to change, or **mutate**, lists using index and slice notation. There are also several list methods that you can use to mutate a list.

List Methods for Adding and Removing Elements

Although you can add and remove elements with index or slice notation, list methods provide a more natural and readable way to mutate a list.

We'll look at several list methods, starting with how to insert a single value into a list at a specified index.

`list.insert()`

The `list.insert()` method is used to insert a single new value into a list. It takes two parameters, an index `i` and a value `x`, and inserts the value `x` at index `i` in the list:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> # Insert "orange" into the second position
>>> colors.insert(1, "orange")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
```

There are a couple of important observations to make about this example.

The first observation applies to all list methods. To use them, you first write the name of the list that you want to manipulate, followed by a dot (`.`), and then the name of the list method.

So, to use `insert()` on the `colors` list, you must write `colors.insert()`. This works just like string and number methods.

Next, notice that when the value "orange" is inserted at the index 1, the value "yellow" and all following values are shifted to the right.

If the value for the index parameter of `.insert()` is larger than the greatest index in the list, then the value is inserted at the end of the list:

```
>>> colors.insert(10, "violet")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'violet']
```

Here the value "violet" is actually inserted at index 5, even though `.insert()` was called with 10 for the index.

You can also use negative indices with `.insert()`:

```
>>> colors.insert(-1, "indigo")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This inserts "indigo" into the slot at index -1, which is the last element of the list. The value "violet" is shifted to the right by one slot.

Important

When you `.insert()` an item into a list, you don't need to assign the result to the original list.

For example, the following code actually erases the colors list:

```
>>> colors = colors.insert(-1, "indigo")
>>> print(colors)
None
```

`.insert()` is said to alter `colors` **in place**. This is true for all list methods that do not return a value.

If you can insert a value at a specified index, then it only makes sense that you can also remove an element at a specified index.

list.pop()

The `list.pop()` method takes one parameter, an index `i`, and removes the value from the list at that index. The value that is removed is returned by the method:

```
>>> color = colors.pop(3)
>>> color
'green'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Here, the value "green" at index 3 is removed and assigned to the variable `color`. When you inspect the `colors` list, you can see that the string "green" has indeed been removed.

Unlike `.insert()`, Python raises an `IndexError` if you pass an argument to `.pop()` that is larger than the last index:

```
>>> colors.pop(10)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    colors.pop(10)
IndexError: pop index out of range
```

Negative indices also work with `.pop()`:

```
>>> colors.pop(-1)
'violet'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

If you don't pass a value to `.pop()`, it removes the last item in the list:

```
>>> colors.pop()
'indigo'
>>> colors
['red', 'orange', 'yellow', 'blue']
```

Removing the final element by calling `.pop()` with no specified index is generally considered the most Pythonic approach.

`list.append()`

The `list.append()` method is used to append a new element to the end of a list:

```
>>> colors.append("indigo")
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

Calling `.append()` increases the length of the list by one and inserts the value "indigo" into the final slot. Note that `.append()` alters the list in place, just like `.insert()`.

Using `.append()` is equivalent to inserting an element at an index greater than or equal to the length of the list. The above example also could have been written as follows:

```
>>> colors.insert(len(colors), "indigo")
```

Using `.append()` is both shorter and more descriptive than using `.insert()` this way, and it's generally considered the more Pythonic way of adding an element to the end of a list.

`list.extend()`

The `list.extend()` method is used to add several new elements to the end of a list:

```
>>> colors.extend(["violet", "ultraviolet"])
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet', 'ultraviolet']
```

`.extend()` takes a single parameter that must be an iterable type. The elements of the iterable are appended to the list in the same order that they appear in the argument passed to `.extend()`.

Just like `.insert()` and `.append()`, `.extend()` alters the list in place.

Typically, the argument passed to `.extend()` is another list, but it could also be a tuple. For example, the above example could be written as follows:

```
>>> colors.extend(("violet", "ultraviolet"))
```

The four list methods discussed in this section make up the most common methods used with lists. The following table serves to recap everything you've seen here:

List Method	Description
<code>.insert(i, x)</code>	Insert the value <code>x</code> at index <code>i</code>
<code>.append(x)</code>	Insert the value <code>x</code> at the end of the list
<code>.extend(iterable)</code>	Insert all the values of <code>iterable</code> in order at the end of the list
<code>.pop(i)</code>	Remove and return the element at index <code>i</code>

In addition to list methods, Python has a couple of useful built-in functions for working with lists of numbers.

Lists of Numbers

One very common operation with lists of numbers is to add up all the values to get the total. You can do this with a `for` loop:

```
>>> nums = [1, 2, 3, 4, 5]
>>> total = 0
>>> for number in nums:
...     total = total + number
...
>>> total
15
```

First, you initialize the variable `total` to 0. Then you loop over each number in `nums` and add it to `total`, finally arriving at the value 15.

Although this `for` loop is straightforward, there's a much more succinct way of doing this in Python:

```
>>> sum([1, 2, 3, 4, 5])  
15
```

The built-in `sum()` function takes a list as an argument and returns the total of all the values in the list.

If the list passed to `sum()` contains any values that aren't numeric, then a `TypeError` is raised:

```
>>> sum([1, 2, 3, "four", 5])  
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>  
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Besides `sum()`, there are two other useful built-in functions for working with lists of numbers: `min()` and `max()`. These functions return the minimum and maximum values in the list, respectively:

```
>>> min([1, 2, 3, 4, 5])  
1  
  
>>> max([1, 2, 3, 4, 5])  
5
```

Note that `sum()`, `min()`, and `max()` also work with tuples:

```
>>> sum((1, 2, 3, 4, 5))  
15  
  
>>> min((1, 2, 3, 4, 5))  
1  
  
>>> max((1, 2, 3, 4, 5))  
5
```

The fact that `sum()`, `min()`, and `max()` are all built into Python tells you that they're used frequently. Chances are, you'll find yourself using them quite a bit in your own programs!

List Comprehensions

Yet another way to create a list from an existing iterable is with a **list comprehension**:

```
>>> numbers = (1, 2, 3, 4, 5)
>>> squares = [num**2 for num in numbers]
>>> squares
[1, 4, 9, 16, 25]
```

A list comprehension is a shorthand for a `for` loop. In the example above, a tuple literal containing five numbers is created and assigned to the `numbers` variable. On the second line, a list comprehension loops over each number in `numbers`, squares each number, and adds it to a new list called `squares`.

Creating the `squares` list using a traditional `for` loop involves creating an empty list, looping over the numbers in `numbers`, and appending the square of each number to the list:

```
>>> squares = []
>>> for num in numbers:
...     squares.append(num**2)
...
>>> squares
[1, 4, 9, 16, 25]
```

List comprehensions are commonly used to convert values in a list to a different type.

For instance, suppose you needed to convert a list of strings containing floating-point values to a list of `float` objects. The following list comprehension achieves this:

```
>>> str_numbers = ["1.5", "2.3", "5.25"]
>>> float_numbers = [float(value) for value in str_numbers]
>>> float_numbers
[1.5, 2.3, 5.25]
```

List comprehensions are not unique to Python, but they are one of its many beloved features. If you find yourself creating an empty list, looping over some other iterable, and appending new items to the list, then chances are you can replace your code with a list comprehension!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a list named `food` with two elements, "rice" and "beans".
2. Append the string "broccoli" to `food` using `.append()`.
3. Add the strings "bread" and "pizza" to `food` using `.extend()`.
4. Print the first two items in the `food` list using `print()` and slice notation.
5. Print the last item in `food` using `print()` and index notation.
6. Create a list called `breakfast` from the string "eggs, fruit, orange juice" using the string `.split()` method.
7. Verify that `breakfast` has three items using `len()`.
8. Create a new list called `lengths` using a list comprehension that contains the lengths of each string in the `breakfast` list.

9.3 Nesting, Copying, and Sorting Tuples and Lists

Now that you've learned what tuples and lists are, how to create them, and how to perform some basic operations with them, let's look at three more concepts:

1. Nesting
2. Copying
3. Sorting

Nesting Lists and Tuples

Lists and tuples can contain values of any type. That means lists and tuples can contain lists and tuples as values. A **nested** list or tuple is a list or tuple that is contained as a value in another list or tuple.

For example, the following list has two values, both of which are other lists:

```
>>> two_by_two = [[1, 2], [3, 4]]  
  
>>> # two_by_two has length 2  
>>> len(two_by_two)  
2  
  
>>> # Both elements of two_by_two are lists  
>>> two_by_two[0]  
[1, 2]  
>>> two_by_two[1]  
[3, 4]
```

Since `two_by_two[1]` returns the list `[3, 4]`, you can use **double index notation** to access an element in the nested list:

```
>>> two_by_two[1][0]  
3
```

First, Python evaluates `two_by_two[1]` and returns `[3, 4]`. Then Python evaluates `[3, 4][0]` and returns the first element, `3`.

In very loose terms, you can think of a list of lists or a tuple of tuples as a sort of table with rows and columns.

The `two_by_two` list has two rows, `[1, 2]` and `[3, 4]`. The columns are made up of the corresponding elements of each row. So the first column contains the elements `1` and `3`, and the second column contains the elements `2` and `4`.

This table analogy is only an informal way of thinking about a list of lists, though. For example, there's no requirement that all the lists in a list of lists have the same length, in which case this table analogy starts to break down.

Copying a List

Sometimes you need to copy one list into another list. However, you can't just reassign one list object to another list object because you'll get this (possibly surprising) result:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals
>>> large_cats.append("Tigger")
>>> animals
['lion', 'tiger', 'frumious Bandersnatch', 'Tigger']
```

In this example, you first assign the list stored in the `animals` variable to the variable `large_cats`, then you add a new string, `"Tigger"`, to the `large_cats` list. But when the contents of `animals` are displayed, you can see that the original list has also been changed.

This is a quirk of object-oriented programming, but it's by design. When you say `large_cats = animals`, the `large_cats` and `animals` variables both refer to the same object.

A variable name is really just a reference to a specific location in computer memory. Instead of copying all the contents of the list object and creating a new list, `large_cats = animals` assigns the memory location referenced by `animals` to `large_cats`. That is, both variables now refer to the same object in memory, and any changes made to one will affect the other.

To get an independent copy of the `animals` list, you can use slice notation to return a new list with the same values:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals[:]
>>> large_cats.append("leopard")
>>> large_cats
['lion', 'tiger', 'frumious Bandersnatch', 'leopard']
>>> animals
["lion", "tiger", "frumious Bandersnatch"]
```

Since no index numbers are specified in the `[:] slice`, every element of the list is returned from beginning to end. The `large_cats` list now has the same elements as `animals`, and in the same order, but you can `.append()` items to it without changing the list assigned to `animals`.

If you want to make a copy of a list of lists, then you can do so using the `[:] notation` you saw earlier:

```
>>> matrix1 = [[1, 2], [3, 4]]
>>> matrix2 = matrix1[:]
>>> matrix2[0] = [5, 6]
>>> matrix2
[[5, 6], [3, 4]]
>>> matrix1
[[1, 2], [3, 4]]
```

Let's see what happens when you change the first element of the second list in `matrix2`:

```
>>> matrix2[1][0] = 1
>>> matrix2
[[5, 6], [1, 4]]
>>> matrix1
[[1, 2], [1, 4]]
```

Notice that the second list in `matrix1` was also altered!

This happens because a list doesn't really contain objects themselves, but references to those objects in memory. `[:]` returns a new list containing the same references as the original list. In programming jargon, this method of copying a list is called a **shallow copy**.

To copy a list and all its elements, you must make what's known as a **deep copy**. A deep copy is a truly independent copy of an object. To make a deep copy of a list, you can use the `deepcopy()` function from Python's `copy` module:

```
>>> import copy
>>> matrix3 = copy.deepcopy(matrix1)
>>> matrix3[1][0] = 3
>>> matrix3
[[5, 6], [3, 4]]
>>> matrix1
[[5, 6], [1, 4]]
```

`matrix3` is created as a deep copy of `matrix1`. When you change an element of `matrix3`, the corresponding element of `matrix1` doesn't change.

Note

For more information on shallow and deep copies, check out [Real Python's “Shallow vs Deep Copying of Python Objects.”](#)

Sorting Lists

Lists have a `.sort()` method that sorts all of the items in ascending order. By default, the list is sorted in alphabetical or numerical order depending on the type of elements in the list:

```
>>> # Lists of strings are sorted alphabetically
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort()
>>> colors
['blue', 'green', 'red', 'yellow']

>>> # Lists of numbers are sorted numerically
>>> numbers = [1, 10, 5, 3]
>>> numbers.sort()
>>> numbers
[1, 3, 5, 10]
```

Notice that `.sort()` sorts the list in place, so you don't need to assign its result to anything.

`.sort()` has an optional parameter called `key` that can be used to adjust how the list gets sorted. The `key` parameter accepts a function, and the list is sorted based on the return value of that function.

For example, to sort a list of strings by the length of each string, you can pass the `len` function to `key`:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort(key=len)
>>> colors
['red', 'blue', 'green', 'yellow']
```

You don't need to call the function when you pass it to `key`. Pass the name of the function without any parentheses. For instance, in the previous example the name `len` is passed to `key`, not `len()`.

Important

The function that gets passed to key must accept only a single argument.

You can also pass user-defined functions to key. In the following example, a function called `get_second_element()` is used to sort a list of tuples by their second elements:

```
>>> def get_second_element(item):
...     return item[1]
...
>>> items = [(4, 1), (1, 2), (-9, 0)]
>>> items.sort(key=get_second_element)
>>> items
[(-9, 0), (4, 1), (1, 2)]
```

Keep in mind that any function that you pass to key must accept only a single argument.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a tuple called `data` with two values. The first value should be the tuple `(1, 2)`, and the second value should be the tuple `(3, 4)`.
2. Write a `for` loop that loops over `data` and prints the sum of each nested tuple. The output should look like this:

```
Row 1 sum: 3
Row 2 sum: 7
```

3. Create the list `[4, 3, 2, 1]` and assign it to the variable `numbers`.
4. Create a copy of the `numbers` list using the `[:] slice notation`.
5. Sort the `numbers` list in numerical order using `.sort()`.

9.4 Challenge: List of lists

Write a program that contains the following lists of lists:

```
universities = [
    ['California Institute of Technology', 2175, 37704],
    ['Harvard', 19627, 39849],
    ['Massachusetts Institute of Technology', 10566, 40732],
    ['Princeton', 7802, 37000],
    ['Rice', 5879, 35551],
    ['Stanford', 19535, 40569],
    ['Yale', 11701, 40500]
]
```

Define a function, `enrollment_stats()`, with a single parameter. This parameter should be a list of lists in which each individual list contains three elements:

1. The name of a university
2. The total number of enrolled students
3. The annual tuition fees

`enrollment_stats()` should return two lists, the first containing all the student enrollment values and the second containing all the tuition fees.

Next, define two functions, `mean()` and `median()`, that take a single list argument and return the mean or median of the values in each list, respectively.

Using `universities`, `enrollment_stats()`, `mean()`, and `median()`, calculate the total number of students, the total tuition, the mean and median numbers of students, and the mean and median tuition values.

Finally, output all values and format the output so that it looks like this:

```
*****
Total students: 77,285
Total tuition: $ 271,905

Student mean:    11,040.71
Student median: 10,566

Tuition mean:   $ 38,843.57
Tuition median: $ 39,849
*****
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

9.5 Challenge: Wax Poetic

In this challenge, you'll write a program that generates poetry.

Create five lists for different word types:

- 1. Nouns:** `["fossil", "horse", "aardvark", "judge", "chef", "mango", "extrovert", "gorilla"]`
- 2. Verbs:** `["kicks", "jingles", "bounces", "slurps", "meows", "explodes", "curdles"]`
- 3. Adjectives:** `["furry", "balding", "incredulous", "fragrant", "exuberant", "glistening"]`
- 4. Prepositions:** `["against", "after", "into", "beneath", "upon", "for", "in", "like", "over", "within"]`
- 5. Adverbs:** `["curiously", "extravagantly", "tantalizingly", "furiously", "sensuously"]`

Randomly select the following number of elements from each list:

- Three nouns
- Three verbs
- Three adjectives
- Two prepositions
- One adverb

You can do this with the `choice()` function in the `random` module. This function takes a list as input and returns a randomly selected element of the list.

For example, here's how you use `random.choice()` to get random element from the list `["a", "b", "c"]`:

```
import random

random_element = random.choice(["a", "b", "c"])
```

Using the randomly selected words, generate and display a poem with the following structure inspired by [Clifford Pickover](#):

```
{A/An} {adj1} {noun1}

{A/An} {adj1} {noun1} {verb1} {prep1} the {adj2} {noun2}
{adverb1}, the {noun1} {verb2}
the {noun2} {verb3} {prep2} a {adj3} {noun3}
```

Here, `adj` stands for adjective and `prep` for preposition.

Here's an example of the kind of poem your program might generate:

```
A furry horse

A furry horse curdles within the fragrant mango
extravagantly, the horse slurps
the mango meows beneath a balding extrovert
```

Each time your program runs, it should generate a new poem.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

9.6 Store Relationships in Dictionaries

One of the most useful data structures in Python is the **dictionary**.

In this section, you'll learn what a dictionary is, how dictionaries differ from lists and tuples, and how to define and use dictionaries in your own code.

What Is a Dictionary?

In plain English, a dictionary is a book containing the definitions of words. Each entry in a dictionary has two parts: the word being defined, and its definition.

Python dictionaries, like lists and tuples, store a collection of objects. However, instead of storing objects in a sequence, dictionaries hold information in pairs of data called **key-value pairs**. That is, each object in a dictionary has two parts: a **key** and a **value**.

The **key** in a key-value pair is a unique name that identifies the **value** part of the pair. Comparing this to an English dictionary, the key is like the word being defined and the value is like the definition of the word.

For example, you could use a dictionary to store the names of states and their capitals:

Key	Value
"California"	"Sacramento"
"New York"	"Albany"
"Texas"	"Austin"

In the table above, the keys of the dictionary are the names of the states, and the values of the dictionary are the names of their capitals.

The difference between an English dictionary and a Python dictionary is that, in Python, the relationship between a key and its value is completely arbitrary. Any key can be assigned to any value.

For example, the following table of key-value pairs is valid:

Key	Value
1	"Sunday"
"red"	12:45pm
17	True

The keys in this table don't appear to be related to the values at all. The only relationship is that each key is assigned to its corresponding value by the dictionary.

In this sense, a Python dictionary is more like a **map** than an English dictionary. The term *map* here refers to a mathematical relation between two sets of values, not a geographical map.

The idea of dictionaries as maps is particularly useful. Under this lens, the English dictionary is a map that relates words to their definitions.

A Python dictionary is a data structure that relates a set of keys to a set of values. Each key is assigned a single value, which defines the relationship between the two sets.

Creating Dictionaries

The following code creates a **dictionary literal** containing names of states and their capitals:

```
>>> capitals = {  
    "California": "Sacramento",  
    "New York": "Albany",  
    "Texas": "Austin",  
}
```

Notice that each key is separated from its value by a colon, each key-value pair is separated by a comma, and the entire dictionary is enclosed in curly braces ({}).

You can also create a dictionary from a sequence of tuples using the built-in `dict()`:

```
>>> key_value_pairs = (  
...     ("California", "Sacramento"),  
...     ("New York", "Albany"),  
...     ("Texas", "Austin"),  
)  
>>> capitals = dict(key_value_pairs)
```

When you inspect a dictionary, it's displayed as a dictionary literal regardless of how it was created:

```
>>> capitals  
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Austin'}
```

Note

If you happen to be following along with a Python version older than 3.6, then you'll notice that the output dictionaries in the interactive window have a different order than the ones that appear in these examples.

Prior to Python 3.6, the order of key-value pairs in a Python dictionary was random. In later versions, the order of the key-value pairs is guaranteed to match the order in which they were inserted.

You can create an empty dictionary using either a literal or a `dict()`:

```
>>> {}
{}
>>> dict()
{}
```

Now that you've created a dictionary, let's look at how you access its values.

Accessing Dictionary Values

To access a value in a dictionary, enclose the corresponding key in square brackets (`[]`) at the end of a dictionary or a variable name assigned to a dictionary:

```
>>> capitals["Texas"]
'Austin'
```

The bracket notation used to access a dictionary value looks similar to the index notation used to get values from strings, lists, and tuples. However, dictionaries are a fundamentally different data structure than sequence types like lists and tuples.

To see the difference, let's step back for a second and notice that we could just as well define the `capitals` dictionary as a list:

```
>>> capitals_list = ["Sacramento", "Albany", "Austin"]
```

You can use index notation to get the capital of each of the three states from the `capitals` dictionary:

```
>>> capitals_list[0] # Capital of California
'Sacramento'

>>> capitals_list[2] # Capital of Texas
'Austin'
```

One nice thing about dictionaries is that they can be used to provide context to the values they contain. Typing `capitals["Texas"]` is easier to understand than `capitals_list[2]`, and you don't have to remember the order of data in a long list or tuple.

This idea of ordering is really the main difference between how items are accessed in a sequence type and in a dictionary.

Values in a sequence type are accessed by index, which is an integer value expressing the order of items in the sequence.

On the other hand, items in a dictionary are accessed by key. Keys don't define an order for dictionary items. They only provide a label that can be used to reference a value.

Adding and Removing Values in a Dictionary

Like lists, dictionaries are mutable data structures. This means you can add and remove items from a dictionary.

Let's add the capital of Colorado to the `capitals` dictionary:

```
>>> capitals["Colorado"] = "Denver"
```

First, you use square bracket notation with "Colorado" as the key, as if you were looking up the value. Then you use the assignment operator (=) to assign the value "Denver" to the new key.

When you inspect `capitals`, you see that a new key, "Colorado", exists with the value "Denver":

```
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Austin',
'Colorado': 'Denver'}
```

Each key in a dictionary can be assigned only a single value. If a key is given a new value, then Python overwrites the old one:

```
>>> capitals["Texas"] = "Houston"  
>>> capitals  
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Houston',  
'Colorado': 'Denver'}
```

To remove an item from a dictionary, use the `del` keyword with the key for the value you want to delete:

```
>>> del capitals["Texas"]  
>>> capitals  
{'California': 'Sacramento', 'New York': 'Albany',  
'Colorado': 'Denver'}
```

Checking the Existence of Dictionary Keys

If you try to access a value in a dictionary using a key that doesn't exist, then Python raises a `KeyError`:

```
>>> capitals["Arizona"]  
Traceback (most recent call last):  
  File "<pyshell#1>", line 1, in <module>  
    capitals["Arizona"]  
KeyError: 'Arizona'
```

The `KeyError` is the most common error encountered when working with dictionaries. Whenever you see it, it means that the program attempted to access a value using a key that doesn't exist.

You can check that a key exists in a dictionary by using the `in` keyword:

```
>>> "Arizona" in capitals  
False  
>>> "California" in capitals  
True
```

With `in`, you can first check that a key exists before doing something with the value for that key:

```
>>> if "Arizona" in capitals:  
...     # Print only if the "Arizona" key exists  
...     print(f"The capital of Arizona is {capitals['Arizona']}")
```

It's important to remember that `in` checks for the existence of keys, not values:

```
>>> "Sacramento" in capitals  
False
```

Even though "Sacramento" is a value for the existing "California" key in `capitals`, checking for its existence returns `False`.

Iterating Over Dictionaries

Like lists and tuples, dictionaries are iterable. But looping over a dictionary is a bit different than looping over a list or a tuple. When you loop over a dictionary with a `for` loop, you iterate over the dictionary's keys:

```
>>> for key in capitals:  
...     print(key)  
...  
California  
New York  
Colorado
```

So, if you want to loop over the `capitals` dictionary and print "The capital of X is Y", where X is the name of the state and Y is the state's capital, then you can do the following:

```
>>> for state in capitals:  
        print(f"The capital of {state} is {capitals[state]}")
```

```
The capital of California is Sacramento  
The capital of New York is Albany  
The capital of Colorado is Denver
```

However, there's a slightly more succinct way to do this using the `.items()` dictionary method, which returns a list-like object containing tuples of key-value pairs. For example, `capitals.items()` returns a list of tuples of states and their corresponding capitals:

```
>>> capitals.items()
dict_items([('California', 'Sacramento'), ('New York', 'Albany'),
('Colorado', 'Denver')])
```

The object returned by `.items()` isn't really a list. It has a special type called a `dict_items`:

```
>>> type(capitals.items())
<class 'dict_items'>
```

Don't worry about what `dict_items` really is. You usually won't work with it directly. The important thing to know is that you can use `.items()` to loop over a dictionary's keys and values simultaneously.

Let's rewrite the previous loop using `.items()`:

```
>>> for state, capital in capitals.items():
...     print(f"The capital of {state} is {capital}")
```

```
The capital of California is Sacramento
The capital of New York is Albany
The capital of Colorado is Denver
```

When you loop over `capitals.items()`, each iteration of the loop produces a tuple containing the state name and the corresponding capital city name. By assigning this tuple to `state, capital`, you ensure that the components are unpacked into the two variables `state` and `capital`.

Dictionary Keys and Immutability

In the `capitals` dictionary that you've been working with throughout this section, each key is a string. However, there's no rule that says dictionary keys must all be of the same type.

For instance, you can add an integer key to capitals:

```
>>> capitals[50] = "Honolulu"
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany',
'Colorado': 'Denver', 50: 'Honolulu'}
```

There's only one restriction on what constitutes a valid dictionary key. Only immutable types are allowed. This means, for example, that a list can't be a dictionary key.

Consider this: What should happen if a list is used as a dictionary key and, later in the code, the list is changed? Should the list be associated with the same value as the old list in the dictionary? Or should the value for the old key be removed from the dictionary altogether?

Rather than make a guess about what should be done, Python raises an exception:

```
>>> capitals[[1, 2, 3]] = "Bad"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

It might not seem fair that some types can be keys and others can't, but it's important for a programming language to have well-defined behavior. It should never guess what the author intends!

For reference, here's a list of all the data types you've learned about so far that are valid dictionary keys:

Valid Dictionary Key Types

integers
floats
strings
Booleans
tuples

Unlike keys, dictionary values can be any valid Python type, including other dictionaries!

Nested Dictionaries

Just as you can nest lists inside other lists and tuples inside other tuples, you can create nested dictionaries. Let's alter the `capitals` dictionary to illustrate this idea:

```
>>> states = {
...     "California": {
...         "capital": "Sacramento",
...         "flower": "California Poppy"
...     },
...     "New York": {
...         "capital": "Albany",
...         "flower": "Rose"
...     },
...     "Texas": {
...         "capital": "Austin",
...         "flower": "Bluebonnet"
...     },
... }
```

Instead of mapping state names to their capital cities, you created a dictionary that maps each state name to a dictionary containing the capital city and the state flower. The value of each key is a dictionary:

```
>>> states["Texas"]
{'capital': 'Austin', 'flower': 'Bluebonnet'}
```

To get the Texas state flower, first get the value at the key "Texas", and then the value at the key "flower":

```
>>> states["Texas"]["flower"]
'Bluebonnet'
```

Nested dictionaries come up more often than you might expect.

They're particularly useful when working with data transmitted over the Web. Nested dictionaries are also great for modeling structured data, such as spreadsheets or relational databases.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create an empty dictionary named `captains`.
2. Using square bracket notation, enter the following data into the dictionary one item at a time:
 - `'Enterprise': 'Picard'`
 - `'Voyager': 'Janeway'`
 - `'Defiant': 'Sisko'`
3. Write two `if` statements that check if `"Enterprise"` and `"Discovery"` exist as keys in the dictionary. Set their values to `"unknown"` if the key does not exist.
4. Write a `for` loop to display the ship and captain names contained in the dictionary. For example, the output should look something like this:

```
The Enterprise is captained by Picard.
```

5. Delete `"Discovery"` from the dictionary.
6. *Bonus:* Make the same dictionary by using `dict()` and passing in the initial values when you first create the dictionary.

9.7 Challenge: Capital City Loop

Review your state capitals along with your knowledge of dictionaries and `while` loops!

First, fill out the following dictionary with the remaining states and their associated capitals in a file called `capitals.py`:

```
capitals_dict = {  
    'Alabama': 'Montgomery',  
    'Alaska': 'Juneau',  
    'Arizona': 'Phoenix',  
    'Arkansas': 'Little Rock',  
    'California': 'Sacramento',  
    'Colorado': 'Denver',  
    'Connecticut': 'Hartford',  
    'Delaware': 'Dover',  
    'Florida': 'Tallahassee',  
    'Georgia': 'Atlanta',  
}
```

Next, pick a random state name from the dictionary and assign both the state and its capital to two variables. You'll need to `import` the `random` module at the top of your program.

Then display the name of the state to the user and ask them to enter the capital. If the user answers incorrectly, then repeatedly ask them for the capital until they either enter the correct answer or type "exit".

If the user answers correctly, then display "Correct" and end the program. However, if the user exits without guessing correctly, display the correct answer and the word "Goodbye".

Note

Make sure the user isn't punished for case sensitivity. In other words, a guess of "Denver" is the same as "denver". Do the same for exiting—"EXIT" and "Exit" should work the same as "exit".

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

9.8 How to Pick a Data Structure

In this chapter, you've learned about three data structures native to Python: lists, tuples, and dictionaries.

You might be wondering, How do I know when to use which data structure? It's a great question, and one many new Python programmers struggle with.

The type of data structure you use depends on the problem you're solving, and there's no hard-and-fast rule you can use every time to pick the right data structure. You'll always need to spend a little time thinking about the problem and which structure works best for it.

Fortunately, there are some guidelines to help you choose.

Use a list when the following are true:

- The data has a natural order.
- You will need to update or alter the data during the program.
- The primary purpose of the data structure is iteration.

Use a tuple when the following are true:

- The data has a natural order.
- You will *not* need to update or alter the data during the program.
- The primary purpose of the data structure is iteration.

Use a dictionary when the following are true:

- The data is unordered, or the order does not matter.
- You will need to update or alter the data during the program.
- The primary purpose of the data structure is looking up values.

9.9 Challenge: Cats With Hats

You have one hundred cats.

One day, you decide to arrange all your cats in a giant circle. Initially, none of your cats has a hat on. You walk around the circle a hundred times, always starting with the first cat (cat #1). Each time you stop at a cat, you check if it has a hat on. If it doesn't, then you put a hat on it. If it does, then you take the hat off.

1. The first round, you stop at every cat, placing a hat on each one.
2. The second round, you stop only at every second cat (#2, #4, #6, #8, and so on).
3. The third round, you stop only at every third cat (#3, #6, #9, #12, and so on).
4. You continue this process until you've made one hundred rounds around the cats. On the last round, you stop only at cat #100.

Write a program that simply outputs which cats have hats at the end.

Note

This is not an easy problem by any means, but the solution is not as complicated as it appears. This problem is often seen in job interviews, as it tests your ability to reason your way through a difficult problem.

Stay calm. Start with a diagram and some pseudo code. Find a pattern, then code!

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

9.10 Summary and Additional Resources

In this chapter, you learned about three data structures: lists, tuples, and dictionaries.

Lists such as [1, 2, 3, 4] are mutable sequences of objects. You can interact with lists using various list methods, including `.append()` and `.extend()`. Lists can be sorted using the `.sort()` method. You can access individual elements of a list using subscript notation, just like strings. Slice notation also works with lists.

Tuples, like lists, are sequences of objects. The big difference between lists and tuples is that tuples are immutable. Once you create a tuple, it can't be changed. Just like lists, you can access tuple elements by index and with slice notation.

Dictionaries store data as key-value pairs. They're not sequences, so you can't access elements by index. Instead, you access elements by their key. Dictionaries are great for storing relationships or when you need quick access to data. Like lists, dictionaries are mutable.

Lists, tuples, and dictionaries are all iterable, meaning they can be looped over. You saw how to loop over all three of these structures using `for` loops.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-tuples-lists-dicts

Additional Resources

To learn more about lists, tuples, and dictionaries, check out the following resources:

- “Lists and Tuples in Python”
- “Dictionaries in Python”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 10

Object-Oriented Programming (OOP)

OOP, or **object-oriented programming**, is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

Conceptually, objects are like the components of a system. Think of a program as a factory assembly line of sorts. At each step of the assembly line a system component processes some material, ultimately transforming raw material into a finished product.

An object contains data, like the raw or preprocessed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

In this chapter, you'll learn how to:

- Create a `class`, which is like a blueprint for creating an object
- Use classes to create new objects
- Model systems with class inheritance

Let's get started!

10.1 Define a Class

Primitive data structures—like numbers, strings, and lists—are designed to represent simple pieces of information, such as the cost of an apple, the name of a poem, or your favorite colors, respectively. What if you want to represent something more complex?

For example, let's say you want to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

First, it can make larger code files more difficult to manage. If you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the element with index 0 is the employee's name?

Second, it can introduce errors if not every employee has the same number of elements in the list. In the `mccoy` list above, the age is missing, so `mccoy[1]` will return "Chief Medical Officer" instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

Classes vs Instances

Classes are used to create user-defined data structures. Classes define functions called **methods**, which identify the behaviors and actions that an object created from the class can perform with its data.

In this chapter, you'll create a `Dog` class that stores some information about the characteristics and behaviors that an individual dog can have.

A class is a blueprint for how something should be defined. It doesn't actually contain any data. The `Dog` class specifies that a name and an age are necessary for defining a dog, but it doesn't contain the name or age of any specific dog.

While the class is the blueprint, an **instance** is an object that is built from a class and contains real data. An instance of the `Dog` class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. An instance is like a form that has been filled out with information. Just like many people can fill out the same form with their own unique information, many instances can be created from a single class.

How to Define a Class

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon. Any code that is indented below the class definition is considered part of the class's body.

Here's an example of a `Dog` class:

```
class Dog:  
    pass
```

The body of the `Dog` class consists of a single statement: the `pass` keyword. `pass` is often used as a placeholder indicating where code will eventually go. It allows you to run this code without Python throwing an error.

Note

Python class names are written in CapitalizedWords notation by convention.

For example, a class for a specific breed of dog like the Jack Russell Terrier would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all `Dog` objects should have. There are a number of properties that we can choose from, including name, age, coat color, and breed. To keep things simple, we'll just use name and age.

The properties that all `Dog` objects must have are defined in a method called `__init__()`. When a new `Dog` object is created, `__init__()` sets the initial **state** of the object by assigning the values of the object's properties. That is, `__init__()` initializes each new instance of the class.

You can give `__init__()` any number of parameters, but the first parameter will always be a variable called `self`. When a new class instance is created, the instance is automatically passed to the `self` parameter in `__init__()` so that new **attributes** can be defined on the object.

Let's update the `Dog` class with an `__init__()` method that creates `.name` and `.age` attributes:

```
class Dog:  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Notice that the `__init__()` method's signature is indented four spaces. The body of the method is indented by eight spaces. This indentation is vitally important. It tells Python that the `__init__()` method belongs to the `Dog` class.

In the body of `__init__()`, there are two statements using the `self` variable:

1. `self.name = name` creates an attribute called `name` and assigns to it the value of the `name` parameter.
2. `self.age = age` creates an attribute called `age` and assigns to it the value of the `age` parameter.

Attributes created in `__init__()` are called **instance attributes**. An instance attribute's value is specific to a particular instance of the class. All `Dog` objects have a name and an age, but the values for the `name` and `age` attributes will vary depending on the `Dog` instance.

On the other hand, **class attributes** are attributes that have the same value for all class instances. You can define a class attribute by assigning a value to a variable name outside of `__init__()`.

For example, the following `Dog` class has a class attribute called `species` with the value "Canis familiaris":

```
class Dog:  
    # Class attribute  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age
```

Class attributes are defined directly beneath the first line of the class name and are indented by four spaces. They must always be assigned an initial value. When an instance of the class is created, class attributes are automatically created and assigned to their initial values.

Use class attributes to define properties that should have the same value for every class instance. Use instance attributes for properties that vary from one instance to another.

Now that we have a `Dog` class, let's create some dogs!

10.2 Instantiate an Object

Open IDLE's interactive window and type the following:

```
>>> class Dog:  
...     pass  
...
```

This creates a new `Dog` class with no attributes or methods.

Creating a new object from a class is called **instantiating** an object. You can instantiate a new `Dog` object by typing the name of the class followed by opening and closing parentheses:

```
>>> Dog()  
<__main__.Dog object at 0x106702d30>
```

You now have a new `Dog` object at `0x106702d30`. This funny-looking string of letters and numbers is a **memory address** that indicates where the `Dog` object is stored in your computer's memory. Note that the address you see on your screen will be different.

Now instantiate a second `Dog` object:

```
>>> Dog()  
<__main__.Dog object at 0x0004ccc90>
```

The new `Dog` instance is located at a different memory address. That's because it's an entirely new instance and is completely unique from the first `Dog` object that you instantiated.

To see this another way, type the following:

```
>>> a = Dog()  
>>> b = Dog()  
>>> a == b  
False
```

In this code, you create two new `Dog` objects and assign them to the variables `a` and `b`. When you compare `a` and `b` using the `==` operator, the result is `False`. Even though `a` and `b` are both instances of the `Dog` class, they represent two distinct objects in memory.

Class and Instance Attributes

Now create a new `Dog` class with a class attribute called `.species` and two instance attributes called `.name` and `.age`:

```
>>> class Dog:  
...     species = "Canis familiaris"  
...     def __init__(self, name, age):  
...         self.name = name  
...         self.age = age  
...  
>>>
```

To instantiate objects of this `Dog` class, you need to provide values for the `name` and `age`. If you don't, then Python raises a `TypeError`:

```
>>> Dog()  
Traceback (most recent call last):  
  File "<pyshell#6>", line 1, in <module>  
    Dog()  
TypeError: __init__() missing 2 required positional  
  arguments: 'name' and 'age'
```

To pass arguments to the `name` and `age` parameters, put values into the parentheses after the class name:

```
>>> buddy = Dog("Buddy", 9)  
>>> miles = Dog("Miles", 4)
```

This creates two new `Dog` instances—one for a nine-year-old dog named Buddy and one for a four-year-old dog named Miles.

The `Dog` class's `__init__()` method has three parameters, so why are

only two arguments passed to it in the example?

When you instantiate a `Dog` object, Python creates a new instance and passes it to the first parameter of `__init__()`. This essentially removes the `self` parameter, so you only need to worry about the `name` and `age` parameters.

After you create the `Dog` instances, you can access their instance attributes using **dot notation**:

```
>>> buddy.name  
'Buddy'  
>>> buddy.age  
9  
  
>>> miles.name  
'Miles'  
>>> miles.age  
4
```

You can access class attributes the same way:

```
>>> buddy.species  
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect. All `Dog` instances have `.species`, `.name`, and `.age` attributes, so you can use those attributes with confidence that they'll always return a value.

Although the attributes are guaranteed to exist, their values *can* be changed dynamically:

```
>>> buddy.age = 10  
>>> buddy.age  
10  
  
>>> miles.species = "Felis silvestris"
```

```
>>> miles.species  
'Felis silvestris'
```

In this example, you change the `.age` attribute of the `buddy` object to 10. Then you change the `.species` attribute of the `miles` object to "Felis silvestris", which is a species of cat. That makes Miles a pretty strange dog, but it is valid Python!

The key takeaway here is that custom objects are mutable by default. Recall that an object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are immutable.

Instance Methods

Instance methods are functions that are defined inside a class and can only be called from an instance of that class. Just like `__init__()`, an instance method's first parameter is always `self`.

Open a new editor window in IDLE and type in the following `Dog` class:

```
class Dog:  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
        # Instance method  
    def description(self):  
        return f"{self.name} is {self.age} years old"  
  
        # Another instance method  
    def speak(self, sound):  
        return f"{self.name} says {sound}"
```

This `Dog` class has two instance methods:

1. `.description()` returns a string displaying the name and age of the dog.
2. `.speak()` has one parameter called `sound` and returns a string containing the dog's name and the sound the dog makes.

Save the modified `Dog` class to a file called `dog.py` and press `F5` to run the program. Then open the interactive window and type the following to see your instance methods in action:

```
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

In the above `Dog` class, `.description()` returns a string containing information about the `Dog` instance `miles`. When writing your own classes, it's a good idea to have a method that returns a string containing useful information about an instance of the class. However, `.description()` isn't the most Pythonic way of doing this.

When you create a `list` object, you can use `print()` to display a string that looks like the list:

```
>>> names = ["David", "Dan", "Joanna", "Fletcher"]
>>> print(names)
['David', 'Dan', 'Joanna', 'Fletcher']
```

Let's see what happens when you `print()` the `miles` object:

```
>>> print(miles)
<__main__.Dog object at 0x00aeff70>
```

When you `print(miles)`, you get a cryptic looking message telling you that `miles` is a `Dog` object at the memory address `0x00aeff70`. This message isn't very helpful. You can change what gets printed by defining a special instance method called `__str__()`.

In the editor window, change the name of the `Dog` class's `.description()` method to `__str__()`:

```
class Dog:  
    # Leave other parts of Dog class as-is  
  
    # Replace .description() with __str__()  
    def __str__(self):  
        return f"{self.name} is {self.age} years old"
```

Save the file and press `F5`. Now, when you `print(miles)`, you get a much friendlier output:

```
>>> miles = Dog("Miles", 4)  
>>> print(miles)  
'Miles is 4 years old'
```

Methods like `__init__()` and `__str__()` are called **dunder methods** because they begin and end with double underscores. There are many dunder methods that you can use to customize classes in Python. Although it's too advanced a topic for a beginning Python book, understanding dunder methods is an important part of mastering OOP in Python.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes. But first, check your understanding with the following review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Modify the `Dog` class to include a third instance attribute called `coat_color`, which stores the color of the dog's coat as a string. Store your new class in a file and test it out by adding the following code at the bottom of the code:

```
philo = Dog("Philo", 5, "brown")
print(f"{philo.name}'s coat is {philo.coat_color}.")
```

The output of your program should be the following:

Philo's coat is brown.

2. Create a `Car` class with two instance attributes: `.color`, which stores the name of the car's color as a string, and `.mileage`, which stores the number of miles on the car as an integer. Then instantiate two `Car` objects—a blue car with 20,000 miles and a red car with 30,000 miles—and print out their colors and mileage. Your output should look like this:

The blue car has 20,000 miles.

The red car has 30,000 miles.

3. Modify the `Car` class with an instance method called `.drive()`, which takes a number as an argument and adds that number to the `.mileage` attribute. Test that your solution works by instantiating a car with 0 miles, then call `.drive(100)` and print the `.mileage` attribute to check that it is set to 100.

10.3 Inherit From Other Classes

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Child classes can override or extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent's attributes and methods but can also specify attributes and methods that are unique to themselves.

Although the analogy isn't perfect, you can think of object inheritance sort of like genetic inheritance.

You may have inherited your hair color from your mother. It's an attribute you were born with. Let's say you decide to color your hair purple. Assuming your mother doesn't have purple hair, you've just **overridden** the hair color attribute that you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you'll also speak English. Now imagine you decide to learn a second language, like German. In this case you've **extended** your attributes because you've added an attribute that your parents don't have.

Dog Park Example

Pretend for a moment that you're at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The `Dog` class that you wrote in the previous section can distinguish dogs by name and age but not by breed.

You could modify the `Dog` class in the editor window by adding a `.breed` attribute:

```
class Dog:  
    species = "Canis familiaris"  
  
    def __init__(self, name, age, breed):  
        self.name = name  
        self.age = age  
        self.breed = breed
```

The instance methods defined earlier are omitted here because they aren't important for this discussion.

Press **F5** to save the file. Now you can model the dog park by instantiating a bunch of different dogs in the interactive window:

```
>>> miles = Dog("Miles", 4, "Jack Russell Terrier")
>>> buddy = Dog("Buddy", 9, "Dachshund")
>>> jack = Dog("Jack", 3, "Bulldog")
>>> jim = Dog("Jim", 5, "Bulldog")
```

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like *woof*, but dachshunds have a higher-pitched bark that sounds more like *yap*.

Using just the `Dog` class, you must supply a string for the `sound` argument of `.speak()` every time you call it on a `Dog` instance:

```
>>> buddy.speak("Yap")
'Buddy says Yap'

>>> jim.speak("Woof")
'Jim says Woof'

>>> jack.speak("Woof")
'Jack says Woof'
```

Passing a string to every call to `.speak()` is repetitive and inconvenient. Moreover, the string representing the sound that each `Dog` instance makes should be determined by its `.breed` attribute, but here you have to manually pass the correct string to `.speak()` every time it's called.

You can simplify the experience of working with the `Dog` class by creating a child class for each breed of dog. This allows you to extend the functionality that each child class inherits, including specifying a default argument for `.speak()`.

Parent Classes vs Child Classes

Let's create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here's the full definition of the `Dog` class:

```
class Dog:  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def __str__(self):  
        return f"{self.name} is {self.age} years old"  
  
    def speak(self, sound):  
        return f"{self.name} says {sound}"
```

To create a child class, you create new class with its own name and then put the name of the parent class in parentheses. The following creates three new child classes of the `Dog` class:

```
class JackRussellTerrier(Dog):  
    pass  
  
class Dachshund(Dog):  
    pass  
  
class Bulldog(Dog):  
    pass
```

With the child classes defined, you can now instantiate some dogs of specific breeds:

```
>>> miles = JackRussellTerrier("Miles", 4)  
>>> buddy = Dachshund("Buddy", 9)  
>>> jack = Bulldog("Jack", 3)  
>>> jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

```
>>> miles.species  
'Canis familiaris'  
  
>>> buddy.name  
'Buddy'  
  
>>> print(jack)  
Jack is 3 years old  
  
>>> jim.speak("Woof")  
'Jim says Woof'
```

You can see which class an object belongs to using the built-in `type()`:

```
>>> type(miles)  
<class '__main__.JackRussellTerrier'>
```

What if you want to determine if `miles` is also an instance of the `Dog` class? You can do this with the built-in `isinstance()`:

```
>>> isinstance(miles, Dog)  
True
```

Notice that `isinstance()` takes two arguments, an object and a class. In the example above, `isinstance()` checks if `miles` is an instance of the `Dog` class and returns `True`.

The `miles`, `buddy`, `jack`, and `jim` objects are all `Dog` instances, but `miles` is not a `Bulldog` instance, and `jack` is not a `Dachshund` instance:

```
>>> isinstance(miles, Bulldog)  
False  
  
>>> isinstance(jack, Dachshund)  
False
```

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've created child classes for some different breeds of dogs, let's give each breed its own sound.

Extend the Functionality of a Parent Class

Since different breeds of dogs have slightly different barks, you want to provide a default value for the `sound` argument of their respective `.speak()` methods. To do this, you need to override `.speak()` in the class definition for each breed.

To override a method defined on the parent class, you define a method with the same name on the child class. Here's what that looks like for the `JackRussellTerrier` class:

```
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

Now `.speak()` is defined on the `JackRussellTerrier` class with the default argument for `sound` set to "Arf".

Update `dog.py` with the new `JackRussellTerrier` class and press `F5` to save and run the file. You can now call `.speak()` on a `JackRussellTerrier` instance without passing an argument to `sound`:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Sometimes dogs make different barks, so if Miles gets angry and growls, you can still call `.speak()` with a different sound:

```
>>> miles.speak("Grrr")
'Miles says Grrr'
```

One thing to keep in mind about class inheritance is that changes to the parent class automatically propagate to child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, in the editor window, change the string returned by `.speak()` in the `Dog` class:

```
class Dog:  
    # Leave other attributes and methods as they are  
  
    # Change the string returned by .speak()  
    def speak(self, sound):  
        return f"{self.name} barks: {sound}"
```

Save the file and press **F5**. Now, when you create a new `Bulldog` instance named `jim`, `jim.speak()` returns the new string:

```
>>> jim = Bulldog("Jim", 5)  
>>> jim.speak("Woof")  
'Jim barks: Woof'
```

However, calling `.speak()` on a `JackRussellTerrier` instance won't show the new style of output:

```
>>> miles = JackRussellTerrier("Miles", 4)  
>>> miles.speak()  
'Miles says Arf'
```

Sometimes it makes sense to completely override a method from a parent class. But in this instance, we don't want the `JackRussellTerrier` class to lose any changes that might be made to the formatting of the output string of `Dog.speak()`.

To do this, you still need to define a `.speak()` method on the child `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the `Dog` class's `.speak()` *inside* the child class's `.speak()` method using the same arguments that you passed to `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using `super()`:

```
class JackRussellTerrier(Dog):  
    def speak(self, sound="Arf"):  
        return super().speak(sound)
```

When you call `super().speak(sound)` inside `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable `sound`.

Update `dog.py` with the new `JackRussellTerrier` class. Save the file and press `F5` so you can test it in the interactive window:

```
>>> miles = JackRussellTerrier("Miles", 4)  
>>> miles.speak()  
'Miles barks: Arf'
```

Now when you call `miles.speak()`, you'll see output reflecting the new formatting in the `Dog` class.

Important

In the above examples, the **class hierarchy** is very simple: the `JackRussellTerrier` class has a single parent class, `Dog`. In real-world examples, the class hierarchy can get quite complicated.

`super()` does much more than just search the parent class for a method or an attribute. It traverses the entire class hierarchy for a matching method or attribute. If you aren't careful, `super()` can have surprising results.

In the next section, you'll bring together everything you've learned by using classes to model a farm. Before you tackle the assignment, check your understanding with the review exercises below.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a `GoldenRetriever` class that inherits from the `Dog` class. Give the `sound` argument of `GoldenRetriever.speak()` a default value of "Bark". Use the following code for your parent `Dog` class:

```
class Dog:  
    species = "Canis familiaris"  
  
    def __init__(self, name, age):  
        self.name = name  
        self.age = age  
  
    def __str__():  
        return f"{self.name} is {self.age} years old"  
  
    def speak(self, sound):  
        return f"{self.name} says {sound}"
```

2. Write a `Rectangle` class that must be instantiated with two attributes: `.length` and `.width`. Add an `.area()` method to the class that returns the area (`length * width`) of the rectangle.

Then write a `Square` class that inherits from the `Rectangle` class and is instantiated with a single attribute called `.side_length`. Test your `Square` class by instantiating a `Square` with a `.side_length` of 4. Calling `.area()` should return 16.

Set the `.width` property of your `Square` instance to 5. Then call `.area()` again. The return value should be 20.

This example illustrates how class inheritance isn't always a good model for subset relationships. In mathematics, all squares are rectangles, but this isn't necessarily true in computer programming.

Be careful to define behaviors so that they reflect expectations, and use class hierarchies with caution.

10.4 Challenge: Model a Farm

In this assignment, you'll create a simplified model of a farm. As you work through this assignment, keep in mind that there are a number of correct answers.

The focus of this assignment is less about the Python class syntax and more about software design in general, which is highly subjective. This assignment is intentionally left open-ended to encourage you to think about how you would organize your code into classes.

Before you write any code, grab a pen and paper and sketch out a model of your farm, identifying classes, attributes, and methods. Think about inheritance. How can you prevent code duplication? Take the time to work through as many iterations as you feel are necessary.

The actual requirements are open to interpretation, but try to adhere to these guidelines:

1. You should have at least four classes: the parent `Animal` class and at least three child animal classes that inherit from `Animal`.
2. Each class should have a few attributes and at least one method that models some behavior appropriate for a specific animal or all animals—walking, running, eating, sleeping, and so on.
3. Keep it simple. Utilize inheritance. Make sure you output details about the animals and their behaviors.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

10.5 Summary and Additional Resources

In this chapter, you learned about object-oriented programming (OOP), which is a programming paradigm used by most modern programming languages, including Java, C#, C++, and Python.

You saw how to define a class, which is a sort of blueprint for an object, and how to instantiate an object from a class. You also learned about attributes, which correspond to the properties of an object, and methods, which correspond to the behaviors and actions of an object.

Finally, you learned how inheritance works by creating child classes from a parent class. You saw how to reference a method on a parent class using `super()` and how to check if an object inherits from another class by using `isinstance()`.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-oop

Additional Resources

You've seen the basics of OOP, but there is so much more to learn! Continue your journey with the following resources:

- [Official Python Documentation](#)
- [OOP Articles on Real Python](#)

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 11

Modules and Packages

As you gain experience writing code, you'll eventually work on projects that are so large that keeping all the code in a single file becomes cumbersome.

Instead of writing a single file, you can put related code into separate files called **modules**. Individual modules can be put together like building blocks to create a larger application.

In this chapter, you'll learn how to:

- Create your own modules
- Use modules in another file via the `import` statement
- Organize several modules into a **package**

Let's get started!

11.1 Working With Modules

A **module** is a file containing Python code that can be reused in other Python code files.

Technically, every Python file that you've created while reading this book is a module, but you haven't seen how to use code from one module inside another.

There are four main advantages to breaking a program into modules:

1. **Simplicity:** Modules are focused on a single problem.
2. **Maintainability:** Small files are better than large files.
3. **Reusability:** Modules reduce duplicate code.
4. **Scoping:** Modules have their own **namespaces**.

In this section, you'll explore modules in more detail. You'll learn how to create them with IDLE, how to import one module into another, and how modules create namespaces.

Creating Modules

Open IDLE and start a new editor window by selecting **File** \gg **New File** or by pressing **Ctrl** + **N**. In the editor window, define a function, `add()`, that returns the sum of its two parameters:

```
# adder.py
def add(x, y):
    return x + y
```

Select **File** \gg **Save** or press **Ctrl** + **S** to save the file as `adder.py` in a new directory on your computer called `myproject/`. `adder.py` is a Python module! It's not a complete program, but not all modules need to be.

Now open another new editor window by pressing **Ctrl** + **N** and type the following code:

```
# main.py
value = add(2, 2)
print(value)
```

Save the file as `main.py` in the same `myproject/` folder you just created. Then press **F5** to run the module.

When the module runs, you'll see a `NameError` displayed in IDLE's interactive window:

```
Traceback (most recent call last):
  File "//Documents/myproject/main.py", line 1, in <module>
    value = add(2, 2)
NameError: name 'add' is not defined
```

It makes sense that a `NameError` occurs because `add()` is defined in `adder.py` and not in `main.py`. In order to use `add()` in `main.py`, you must first import the `adder` module.

Importing One Module Into Another

In the editor window for `main.py`, add the following line to the top of the file:

```
# main.py
import adder # <-- Add this line

# Leave the code below unchanged
value = add(2, 2)
print(value)
```

When you `import` one module into another, the contents of the imported module become available in the other. The module with the `import` statement is called the **calling module**. In this example, `adder.py` is the imported module and `main.py` is the calling module.

Press **Ctrl**+**S** to save `main.py`, then press **F5** to run the module. The `NameError` exception is still raised. That's because `add()` can only be accessed from the `adder` namespace.

A **namespace** is a collection of names, such as variable names, function names, and class names. Every Python module has its own namespace.

Variables, functions, and classes in a module can be accessed from within the same module by just typing their name. That's how you've been doing it throughout this book so far. However, this doesn't work for imported modules.

To access a name in an imported module from the calling module, type the imported module's name followed by a dot (.) and the name you want to use:

```
<module>.<name>
```

For instance, to use `add()` in the `adder` module, you need to type `adder.add()`.

Important

The name used to import a module is the same as the module's filename.

For this reason, module filenames must be valid Python identifiers. That means they may only contain uppercase and lowercase letters, numbers, and underscores (_), and they may not start with a digit.

Update the code in `main.py` as follows:

```
# main.py
import adder

value = adder.add(2, 2) # <-- Change this line
print(value)
```

Save the file and run the module. The value 4 is printed in the interactive window.

When you type `import <module>` at the beginning of a file, the module's entire namespace is imported. Any new variables or functions added to `adder.py` will be accessible in `main.py` without you having to import anything new.

Open the editor window for `adder.py` and add the following function below `add()`:

```
# adder.py
# Leave this code unchanged
def add(x, y):
    return x + y

def double(x): # <-- Add this function
    return x + x
```

Save the file. Then open the editor window for `main.py` and add the following code:

```
# main.py
import adder

value = adder.add(2, 2)
double_value = adder.double(value) # <-- Add this line
print(double_value) # <-- Change this line
```

Now save and run `main.py`. When the module runs, the value 8 is displayed in the interactive window. Since `double()` already exists in the `adder` namespace, no `NameError` is raised.

Import Statement Variations

The `import` statement is flexible. There are two variations that you should know about:

1. `import <module> as <other_name>`
2. `from <module> import <name>`

Let's look at each of these variations in detail.

```
import <module> as <other_name>
```

You can change the name of an import using the `as` keyword:

```
import <module> as <other_name>
```

When you import a module this way, the module's namespace is accessed through `<other_name>` instead of `<module>`.

For example, change the `import` statement in `main.py` to the following:

```
import adder as a # <-- Change this line

# Leave the code below unchanged
value = adder.add(2, 2)
double_value = adder.double(value)
print(double_value)
```

Save the file and press `F5`. A `NameError` is raised:

```
Traceback (most recent call last):
  File "//Mac/Home/Documents/myproject/main.py", line 3, in <module>
    value = adder.add(2, 2)
NameError: name 'adder' is not defined
```

The adder name is no longer recognized because the module has been imported with the name a instead of adder.

To make `main.py` work, you need to replace `adder.add()` and `adder.double()` with `a.add()` and `a.double()`:

```
import adder as a

value = a.add(2, 2) # <-- Change this line
double_value = a.double(value) # <-- Change this line, too
print(double_value)
```

Now save the file and run the module. No `NameError` is raised, and the value 8 is displayed in the interactive window.

```
from <module> import <name>
```

Instead of importing the entire namespace, you can import only a specific name from a module. To do this, replace the `import` statement with the following:

```
from <module> import <name>
```

For example, in `main.py`, change the `import` statement to the following:

```
# main.py
from adder import add # <-- Change this line

value = adder.add(2, 2)
double_value = adder.double(2, 2)
print(double_value)
```

Save the file and press `F5`. A `NameError` exception is raised:

```
Traceback (most recent call last):
File "//Documents/myproject/main.py", line 3, in <module>
    value = adder.add(2, 2)
NameError: name 'adder' is not defined
```

The above traceback tells you that the name adder is undefined. Only the name add is imported from adder.py and is placed in the main.py module's local namespace. That means you can use add() without having to type adder.add().

Replace adder.add() and adder.double() in main.py with add() and double():

```
# main.py
from adder import add

value = add(2, 2) # <-- Change this line
double_value = double(value) # <-- Change this line, too
print(double_value)
```

Now save the file and run the module. What do you think happens?

Another NameError is raised:

```
Traceback (most recent call last):
  File "//Documents/myproject/main.py", line 4, in <module>
    double_value = double(value)
NameError: name 'double' is not defined
```

This time, the NameError tells you that the name double isn't defined, which proves that only the add name was imported from the adder module.

You can import the double name by adding it to the import statement in main.py:

```
# main.py
from adder import add, double # <-- Change this line

# Leave the code below unchanged
value = add(2, 2)
double_value = double(value)
print(double_value)
```

Save and run the module. Now the module runs without producing a `NameError`. The value 8 is displayed in the interactive window.

Summary of Import Statements

The following table summarizes what you've learned about importing modules:

Import Statement	Result
<code>import <module></code>	Import the entire namespace of <code><module></code> into the name <code><module></code> . Import module names can be accessed from the calling module with <code><module>.name</code> .
<code>import <module> as <other_name></code>	Import the entire namespace of <code><modules></code> into the name <code><other_name></code> . Import module names can be accessed from the calling module with <code><other_name>.name</code> .
<code>from <module> import <name1>, <name2>, ...</code>	Import only the names <code><name1></code> , <code><name2></code> , etc., from <code><module></code> . The names are added to the calling module's local namespace and can be accessed directly.

Separate namespaces are one of the great advantages of dividing code into individual modules, so let's take some time to explore why namespaces matter and why you should care about them.

Why Use Namespaces?

Suppose every person on the entire planet will be given an ID number. In order to distinguish one person from the next, each ID number needs to be unique. We'll need a whole bunch of ID numbers to make that work!

The world is divided into countries, so we can group people by their

country of birth. If we assign each country a unique code, we can attach that code to a person's ID number. For example, a person from the United States might have an ID of `us-357`, and a person from Great Britain might have an ID of `gb-246`.

Two people from different countries can now have the same ID number. We can distinguish them because their IDs begin with different country codes. Each person from the same country must have a unique ID number, but we no longer need globally unique IDs.

The country codes in this scenario are an example of **namespaces**, and they illustrate three of the main reasons for using namespaces:

1. They **group** names into logical containers.
2. They **prevent clashes** between duplicate names.
3. They **provide context** to names.

Namespaces in code provide the same advantages.

You've seen three different ways to import one module into another. Keeping in mind the advantages that namespaces provide can help you determine which kind of `import` statement makes the most sense.

In general, `import <module>` is the preferred approach because it keeps the imported module's namespace completely separate from the calling module's namespace. Moreover, every name from the imported module is accessed from the calling module with the `<module>.name` format, which immediately tells you in which module the name originates.

There are two reasons you might use the `import <module> as <other_name>` format:

1. The module name is long, and you wish to import an abbreviated version of it.
2. The module name clashes with an existing name in the calling module.

The statement `import <module> as <other_name>` still keeps the imported module's namespace separate from the calling module's namespace. The tradeoff is that the name you give the module might not be as easily recognizable as the original module name.

Importing specific names from a module is generally the least preferred way to import code from a module. The imported names are added directly to the calling module's namespace, completely removing them from context of the calling module.

Sometimes, modules contain a single function or class that has the same name as the module. For example, there is a module in the Python standard library called `datetime` that contains a class called `datetime`.

Suppose you add the following `import` statement to your code:

```
import datetime
```

This imports the `datetime` module into your code's namespace, so to use the `datetime` class contained in the `datetime` module, you need to type the following:

```
datetime.datetime(2020, 2, 2)
```

Don't worry about how the `datetime` class works right now. The important part of this example is that having to constantly type `datetime.datetime` whenever you want to use the `datetime` class is redundant and tiresome.

This is a great example of when it's appropriate to use one of the variations of the `import` statement. To keep the context of the `datetime` package, it's common for Python programmers to import the package and rename it as `dt`:

```
import datetime as dt
```

Now, to use the `datetime` class, you only need to type `dt.datetime`:

```
dt.datetime(2020, 2, 2)
```

It's also common to import the `datetime` class directly into the calling module's namespace:

```
from datetime import datetime
```

This is fine because the context isn't really lost. The class and the module share the same name, after all.

When you import the `datetime` class directly, you no longer have to use dotted module names to access it:

```
datetime(2020, 2, 2)
```

The various import statements allow you to minimize the time you spend typing unnecessarily long dotted module names. That said, abusing the various `import` statements can lead to a loss of context, resulting in code that is more difficult to understand.

Always use good judgment when importing modules so that you preserve the most context possible.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a module called `greeter.py` that contains a single function `greet()`. This function should accept a single string parameter `name` and print the text `Hello {name}!` to the interactive window, with `{name}` replaced by the function argument.
2. Create a module called `main.py` that imports `greet()` from `greeter.py` and calls the function with the argument "Real Python".

11.2 Working With Packages

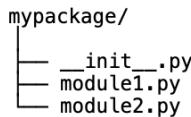
Modules allow you to divide a program into individual files that can be reused as needed. Related code can be organized into a single module and kept separate from other code.

Packages take this organizational structure one step further by allowing you to group related modules under a single namespace.

In this section, you'll learn how to create your own Python package and import code from that package into another module.

Creating Packages

A **package** is a folder that contains one or more Python modules. It must also contain a special module called `__init__.py`. Here's an example of a package to illustrate this structure:



The `__init__.py` module doesn't need to contain any code! It only needs to exist so that Python recognizes the `my/package/` folder as a Python package.

Using your computer's file explorer or whatever tool you're comfortable with, create a new folder somewhere on your computer called `packages_example/`. Inside that folder, create another folder called `my/package/`.

The `packages_example/` folder is called the **project folder**, or **project root folder**, because it contains all the files or folders in the `packages_examples` project. The `my/package/` folder will eventually become a Python package. It isn't one right now because it doesn't contain any modules.

Open IDLE and create a new editor window by pressing **Ctrl** + **N**. At the top of the file, add the following comment:

```
# main.py
```

Now press **Ctrl** + **S** and save the file as `main.py` in the `packages-example/` folder you created earlier.

Open another editor window by pressing **Ctrl** + **N**. Insert the following at the top of the file:

```
# __init__.py
```

Then save the file as `__init__.py` in the `mypackage/` subfolder of your `packages_example` folder.

Finally, create two more editor windows. Save these files as `module1.py` and `module2.py`, respectively, in your `mypackage/` folder, and insert comments at the top of each file containing the file name.

When you're done, you should have five IDLE windows open: the interactive window and four editor windows.

Now that we've created the package structure, let's add some code. In the `module1.py` file, add the following function:

```
# module1.py
def greet(name):
    print(f"Hello, {name}!")
```

In the `module2.py` file add the following:

```
# module2.py
def depart(name):
    print(f"Goodbye, {name}!")
```

Make sure you save both of the `module1.py` and `module2.py` files! You're now ready to import and use these modules in the `main.py` module.

Importing Modules From Packages

In your `main.py` file, add the following code:

```
# main.py
import mypackage

mypackage.module1.greet("Pythonista")
mypackage.module2.depart("Pythonista")
```

Save `main.py` and press `F5` to run the module. In the interactive window, an `AttributeError` is raised:

```
Traceback (most recent call last):
  File "\MacHome\Documents\spackages_exampemain.py", line 5, in <module>
    mypackage.module1.greet("Pythonista")
AttributeError: module 'mypackage' has no attribute 'module1'
```

When you import the `mypackage` module, the `module1` and `module2` namespaces aren't imported automatically—you need to import them, too.

Change the import statement at the top of `main.py`:

```
# main.py
import mypackage.module1 # <-- Change this line

# Leave the below code unchanged
mypackage.module1.greet("Pythonista")
mypackage.module2.depart("Pythonista")
```

Now save and run the `main.py` module. You should see the following output in the interactive window:

```
Hello, Pythonista!
Traceback (most recent call last):
  File "\MacHome\Documents\spackages_exampemain.py", line 6, in <module>
    mypackage.module2.depart("Pythonista")
AttributeError: module 'mypackage' has no attribute 'module2'
```

You can tell that `mypackage.module1.greet()` was called because `Hello, Pythonista!` is displayed in the interactive window.

However, `mypackage.module2.depart()` was not called. That line raised an attribute error because the only module imported from `mypackage` so far is `module1`.

To import `module2`, add the following `import` statement to the top of your `main.py` file:

```
# main.py
import mypackage.module1
import mypackage.module2 # <-- Add this line

# Leave the below code unchanged
mypackage.module1.greet("Pythonista")
mypackage.module2.depart("Pythonista")
```

Now when you save and run `main.py`, both `greet()` and `depart()` get called:

```
Hello, Pythonista!
Goodbye, Pythonista!
```

In general, modules are imported from packages using **dotted module names** with the following format:

```
import <package_name>.<module_name>
```

First, type the name of the package, followed by a dot (.) and the name of the module you want to import.

Important

Just like module file names, package folder names must be valid Python identifiers. They may contain only uppercase and lowercase letters, numbers, and underscores (_), and they may not start with a digit.

As with modules, there are several variations on the `import` statement that you can use when importing packages.

Import Statement Variations for Packages

There are three variations of the `import` statement that you learned for importing names from modules. These three variations translate to the following four variations for importing modules from packages:

1. `import <package>`
2. `import <package> as <other_name>`
3. `from <package> import <module>`
4. `from <package> import <module> as <other_name>`

These variations work much the same as their counterparts.

For instance, instead of importing `mypackage.module1` and `mypackage.module2` on separate lines, you can import both on the same line. Change your `main.py` file to the following:

```
# main.py
from mypackage import module1, module2

module1.greet("Pythonista")
module2.depart("Pythonista")
```

When you save and run the module, the same output as before is displayed in the interactive window.

You can change the name of an imported module using the `as` keyword:

```
# main.py
from mypackage import module1 as m1, module2 as m2

m1.greet("Pythonista")
m2.depart("Pythonista")
```

You can also import individual names from a package module. For instance, you can rewrite `main.py` to the following without changing what gets printed when you save and run the module:

```
# main.py
from mypackage.module1 import greet
from mypackage.module2 import depart

greet("Pythonista")
depart("Pythonista")
```

With so many ways to import packages, it's natural to wonder which way is best.

Guidelines for Importing Packages

The same guidelines for importing names from modules apply to importing modules from packages. You should prefer that imports be as explicit as possible so that the modules and names imported into the calling module have the appropriate context.

In general, the following format is the most explicit:

```
import <package>.<module>
```

To access names from module, type something like the following:

```
<package>.<module>.<name>
```

This way, when you encounter names from the imported module, there's no question where those names come from. But sometimes package and module names are long, and you find yourself typing `<package>.<module>` over and over again in your code.

The following format allows you to skip the package name and import just the module name into the calling module's namespace:

```
from <package> import <module>
```

Now you can just type `<module>. <name>` to access a name from the module. While this no longer tells you which package the name comes from, it does keep the context of the module apparent.

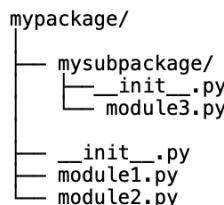
Finally, the following format is generally ambiguous and should only be used when there's no risk of importing a name from a module that clashes with a name in the calling module:

```
from <package>. <module> import <name>
```

Now that you've seen how to import modules from packages, let's take a quick look at how to nest packages inside other packages.

Importing Modules From Subpackages

A package is just a folder containing one or more Python modules, one of which must be named `__init__.py`. So it's entirely possible to have the following package structure:



A package nested inside another package is called a **subpackage**. For example, the `mysubpackage/` folder is a subpackage of `mypackage` because it contains an `__init__.py` module as well as a second module called `module3.py`.

Using your computer's file explorer or some other tool, create the `mysubpackage/` folder on your computer. Make sure you place the folder inside the `mypackage/` folder you created earlier.

In IDLE, open two new editor windows. Create the files `__init__.py` and `module3.py` and save both modules to the `mysubpackage/` folder.

In your `module3.py` file, add the following code:

```
# module3.py

people = ["John", "Paul", "George", "Ringo"]
```

Now open the `main.py` file in your root `packages_examples/` project folder. Remove any existing code and replace it with the following:

```
# main.py
from mypackage.module1 import greet
from mypackage.mysubpackage.module3 import people

for person in people:
    greet(person)
```

The `people` list from the `module3` module inside `mysubpackage` is imported via the dotted module name `mypackage.mysubpackage.module3`.

Now save and run `main.py`. The following output is displayed in the interactive window:

```
Hello, John!
Hello, Paul!
Hello, George!
Hello, Ringo!
```

Subpackages are great for organizing code inside very large packages. They help keep the folder structure of a package clean and organized.

However, deeply nested subpackages introduce long dotted module names. You can imagine how much typing it would take to import a module from a subpackage of a subpackage of a subpackage of a package.

It's good practice to keep your subpackages at most one or two levels deep.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. In a new project folder called `package_exercises/`, create a package called `helpers` with three modules: `__init__.py`, `string.py`, and `math.py`.

In the `string.py` module, add a function called `shout()` that takes a single string parameter and returns a new string with all of the letters in uppercase.

In the `math.py` module, add a function called `area()` that takes two parameters called `length` and `width` and returns their product `length * width`.

2. In the root project folder, create a module called `main.py` that imports the `shout()` and `area()` functions. Use `shout()` and `area()` to print the following output:

```
THE AREA OF A 5-BY-8 RECTANGLE IS 40
```

11.3 Summary and Additional Resources

In this chapter, you learned how to create your own Python modules and packages and how to import objects from one module into another. You saw that dividing code into modules and packages is advantageous for the following reasons:

- Small code files are **simpler** than large code files.
- Small code files are **easier to maintain** than large code files.
- Modules can be **reused** throughout a project.
- Modules group related objects into isolated **namespaces**.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-modules-packages

Additional Resources

To learn more about modules and packages, check out the following resources:

- [Python Modules and Packages \(Course\)](#)
- [Absolute vs Relative Imports \(Course\)](#)

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 12

File Input and Output

So far, you've written programs that get their input from one of two sources: the user or the program itself. Program output has been limited to displaying some text in IDLE's interactive window.

These input and output methods aren't useful in several common scenarios, such as when:

- The input values are unknown while writing the program
- The program requires more data than a user can be expected to type in by themselves
- Output must be shared with others after the program runs

This is where files come in.

In this chapter, you'll learn how to:

- Work with file paths and file metadata
- How to read and write text files
- How to read and write **comma-separated value (CSV)** files
- How to create, delete, copy, and move files and folders

Let's dive in!

12.1 Files and the File System

You've likely been working with computer files for a long time. Even so, there are some things that programmers need to know about files that general users do not.

In this section, you'll learn the concepts necessary to get started working with files in Python.

The Anatomy of a File

There are many types of files, including text files, image files, audio files, and PDF files. Regardless of a file's type, though, a file is composed of a sequence of **bytes** called the **file contents**.

A byte is an integer with a value between 0 and 255. Bytes are the values that are stored on a physical storage device when a file is saved. When you access a file on a computer, the bytes in the file are read in sequence from the disk.

There's nothing about the file itself that dictates how to interpret its contents. As a programmer, you're responsible for converting bytes to the appropriate format when your program opens a file. This might sound difficult, but Python does a lot of the hard work for you.

For example, Python can convert the numerical bytes of a text file into text characters for you. You don't need to know exactly how this conversion works. There are tools in the standard library for working with all sorts of file types, including images and audio files.

To access a file from a storage device, you need to know which device the file is stored on, how to interact with that device, and where exactly on the device the file is located. This monumental task is managed by a **file system**.

The File System

A computer's file system does two things:

1. It provides an abstract representation of the files stored on the computer and any devices connected to it.
2. It interfaces with devices to control the storage and retrieval of file data.

Python interacts with the file system on your computer and is limited to whatever actions your file system allows.

Important

Different operating systems use different file systems. This is important to keep in mind when writing code that will be run on different operating systems.

The file system manages communication between the computer and the physical storage device. That's good news! It means that, as a Python programmer, you don't need to worry about things like accessing physical storage or spinning a hard disk.

The File System Hierarchy

File systems organize files in a hierarchy of **directories**, which are also known as **folders**. At the top of the hierarchy is a directory called the **root directory**. All other files and directories in the file system are contained in the root directory.

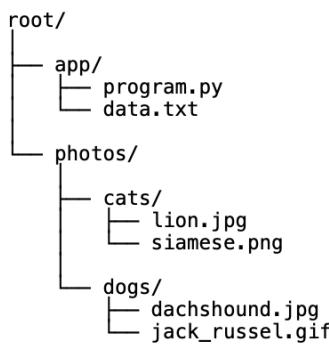
Important

On Windows, every disk drive has its own file hierarchy with the root directory represented by the filename.

macOS and Linux are different in that each drive is represented as a subdirectory of a single root directory.

Every file has a **filename** that must be unique from any other file in the same directory. Directories can also contain other directories called **subdirectories** or **subfolders**.

The following **directory tree** visualizes the hierarchy of files and directories in an example file system:



In this file system, the root folder is called `root/`. It has two subdirectories called `app/` and `photos/`. The `app/` subdirectory contains a `program.py` file and a `data.txt` file. The `photos/` directory has two subdirectories called `cats/` and `dogs/` that both contain two image files.

File Paths

To locate a file in a file system, you can list the directories in order, starting with the root directory, followed by the name of the file. A string with the file location represented in this manner is called a **file path**.

For example, the file path for the `jack_russel.gif` file in the above file system is `root/photos/dogs/jack_russel.gif`.

How you write file paths depends on your operating system. Here are three examples of file paths on Windows, macOS, and Linux:

1. **Windows:** `C:\Users\David\Documents\hello.txt`

2. **macOS:** /Users/David/Documents/hello.txt
3. **Ubuntu Linux:** /home/David/Documents/hello.txt

All three of these file paths locate a text file named `hello.txt` that is stored in the `Documents` subfolder of the user directory for a user named `David`. As you can see, there are some pretty big differences between file paths from one operating system to another.

On macOS and Ubuntu Linux, the operating system uses a **virtual file system** that organizes all files and directories for all devices on the system under a single root directory, usually represented by a forward slash symbol (`/`). Files and folders from external storage devices are usually located in a subdirectory called `media/`.

On Windows, there is no universal root directory. Each device has a separate file system with a unique root directory that is named with a **drive letter** followed by a colon (`:`) and a backslash symbol (`\`). Typically, the hard drive on which the operating system is installed is assigned the letter `C`, so the root directory of the file system for that drive is `c:\`.

The other major difference between Windows, macOS, and Ubuntu file paths is that directories in a Windows file path are separated by backslashes (`\`), whereas directories in macOS and Ubuntu file paths are separated by forward slashes (`/`).

When you write programs that need to run on multiple operating systems, it's critical that you handle the differences in file paths appropriately. In versions of Python greater than 3.4, the standard library contains a module called `pathlib` that helps take the pain out of handling file paths across operating systems.

12.2 Working With File Paths in Python

Python's `pathlib` module is the main interface for working with file paths. You'll need to import the module before you can do anything with it.

Open IDLE's interactive window and type the following to import `pathlib`:

```
>>> import pathlib
```

The `pathlib` module contains a class called `Path` that is used to represent a file path.

Creating Path Objects

There are several ways to create a new `Path` object:

1. From a string
2. With the `Path.home()` and `Path.cwd()` class methods
3. With the `/` operator

The most straightforward way to create a `Path` object is from a string.

Creating Path Objects From Strings

For instance, the following creates a `Path` object representing the macOS file path `"/Users/David/Documents/hello.txt"`:

```
>>> path = pathlib.Path("/Users/David/Documents/hello.txt")
```

There's a problem, though, with Windows paths. On Windows, directories are separated by backslashes (`\`). Python interprets backslashes as the start of an **escape sequence** that represent a special character in the string, such as the newline character (`\n`).

Attempting to create a `Path` object with the Windows file path `"C:\Users\David\Desktop\hello.txt"` raises an exception:

```
>>> path = pathlib.Path("C:\Users\David\Desktop\hello.txt")
SyntaxError: (unicode error) 'unicodedeescape' codec can't decode bytes
in position 2-3: truncated \UXXXXXXXXX escape
```

There are two ways to get around this problem.

First, you can use a forward slash (/) instead of a backslash (\) in your Windows file paths:

```
>>> path = pathlib.Path("C:/Users/David/Desktop/hello.txt")
```

Python can interpret this just fine and will translate the path appropriately and automatically when interfacing with the Windows operating system.

Second, you can turn the string into a `raw` string by prefixing it with an `r`:

```
>>> path = pathlib.Path(r"C:\Users\David\Desktop\hello.txt")
```

This tells Python to ignore any escape sequences and just read the string as is.

Using `Path.home()` and `Path.cwd()`

Besides creating a `Path` object from a string, the `Path` class has class methods that return the `Path` objects of special directories. Two of the most useful class methods are `Path.home()` and `Path.cwd()`.

Every operating system has a special directory for storing data for the currently logged-in user. This directory is called the user's **home directory**. The location of this directory depends on the operating system:

- **Windows:** `C:\Users\<username>`
- **macOS:** `/Users/<username>`
- **Ubuntu Linux:** `/home/<username>`

The `Path.home()` class method creates a `Path` object representing the home directory regardless of which operating system the code runs on:

```
>>> home = pathlib.Path.home()
```

When you inspect the `home` variable on Windows, you'll see something like this:

```
>>> home  
WindowsPath("C:/Users/David")
```

The `Path` object created is a subclass of `Path` called `WindowsPath`. On other operating systems, the `Path` object returned is a subclass called `PosixPath`.

For example, on macOS, inspecting `home` will display something like the following:

```
>>> home  
PosixPath("/Users/David")
```

For the rest of this section, `WindowsPath` objects will be shown in the example output. However, all of the examples will work with `PosixPath` objects.

Note

`WindowsPath` and `PosixPath` objects share the same methods and attributes. From a programming standpoint, there's no difference between the two types of `Path` objects.

The `Path.cwd()` class method returns a `Path` object representing the **current working directory**, or **CWD**. The current working directory is a dynamic reference to a directory that depends on where a process on the computer is currently working. It always represents your current location in the file system.

When you run IDLE, the current working directory is usually set to the `Documents` directory in the current user's home directory:

```
>>> pathlib.Path.cwd()  
WindowsPath("C:/Users/David/Documents")
```

This is not always the case, though. Moreover, the current working directory may change during the lifetime of a program.

`Path.cwd()` is useful, but be careful when you use it. When you do, make sure you know exactly which directory the current working directory refers to.

Using the / Operator

If you have an existing `Path` object, then you can use the `/` operator to extend the path with subdirectories or filenames.

For example, the following creates a `Path` object representing a file named `hello.txt` in the `Desktop` subdirectory of the current user's home directory:

```
>>> home / "Desktop" / "hello.txt"
WindowsPath('C:/Users/David/Desktop/hello.txt')
```

The `/` operator must always have a `Path` object on the left-hand side. The right-hand side can have a string representing a single file or directory, or it can have a string representing a path or other `Path` object.

Absolute vs Relative Paths

A path that begins with the root directory in a file system is called an **absolute file path**. Not all file paths are absolute. A file path that is not absolute is called a **relative file path**.

Here's an example of a `Path` object that references a relative path:

```
>>> path = pathlib.Path("Photos/image.jpg")
```

Notice that the path string does not start with `c:\` or `/`.

You can determine if a file path is absolute using `.is_absolute()`:

```
>>> path.is_absolute()
False
```

Relative paths only make sense when considered within the context of some other directory. They are perhaps most commonly used to describe the path to a file relative to the current working directory or the user's home directory.

You can extend a relative path to an absolute path using the forward slash (/) operator:

```
>>> home = pathlib.Path.home()  
>>> home / pathlib.Path("Photos/image.png")  
WindowsPath('C:/Users/David/Photos/image.png')
```

On the left of the forward slash (/), put an absolute path to the directory that contains the relative path. Then put the relative path on the right side of the forward slash.

You won't always know how to construct an absolute path, though. In those cases, you can use `Path.resolve()`.

When you call `.resolve()` on an existing `Path` object, a new `Path` object representing the absolute path is returned:

```
>>> relative_path = pathlib.Path("/Users/David")  
>>> absolute_path = relative_path.resolve()  
>>> absolute_path  
WindowsPath('C:/Users/David')
```

`Path.resolve()` attempts to create as much of the absolute path as possible.

Sometimes the relative path is ambiguous. In that case, `.resolve()` returns the relative path. In other words, `.resolve()` isn't guaranteed to return an absolute path.

Once you create a `Path` object, you can inspect the various components of the file path that it refers to.

Accessing File Path Components

All file paths contain a list of directories. The `.parents` attribute of a `Path` object returns an iterable containing a list of directories in the file path:

```
>>> path = pathlib.Path.home() / "hello.txt"
>>> path
WindowsPath("C:/Users/David")
>>> list(path.parents)
[WindowsPath("C:/Users/David"), WindowsPath("C:/Users"),
WindowsPath("C:/")]
```

Notice that the directories are returned in reverse order of how they appear in the file path. That is, the last directory in the path is the first directory in the list of parent directories.

You can iterate over the parent directories in a `for` loop:

```
>>> for directory in path.parents:
...     print(directory)
...
C:\Users\David
C:\Users
C:\
```

The `.parent` attribute returns the name of the first parent directory in the file path as a string:

```
>>> path.parent
WindowsPath('C:/Users/David')
```

`.parent` is a shortcut for `.parents[0]`.

If the file path is absolute, then you can access the root directory of the file path with the `.anchor` attribute:

```
>>> path.anchor
'C:\'
```

Note that `.anchor` returns a string, not another `Path` object.

For relative paths, `.anchor` returns an empty string:

```
>>> path = pathlib.Path("hello.txt")
>>> path.anchor
''
```

The `.name` attribute returns the name of the file or directory that the path points to:

```
>>> home = pathlib.Path.home()      # C:\Users\David
>>> home.name
'David'
>>> path = home / "hello.txt"
>>> path.name
'hello.txt'
```

File names are broken down into two parts. The part to the left of the dot (.) is called the **stem**, and the part to the right of the dot is called the **suffix** or **file extension**.

The `.stem` and `.suffix` attributes return strings containing each of these parts of the filename:

```
>>> path.stem
'hello'
>>> path.suffix
'.txt'
```

You might be wondering at this point how to actually *do* something with the `hello.txt` file. You'll learn how to read and write files in the next section. But before you open a file for reading, it's a good idea to know whether or not that file exists.

Checking Whether a File Path Exists

You can create a `Path` object for a file path even if that path doesn't actually exist. Of course, file paths that don't represent actual files

or directories aren't very useful unless you plan on creating them at some point.

Path objects have an `.exists()` method that returns `True` or `False` depending on whether or not the file path exists on the machine executing the program.

For instance, if you don't have a `hello.txt` file in your home directory, then calling `.exists()` on the Path object representing that file path returns `False`:

```
>>> path = pathlib.Path.home() / "hello.txt"
>>> path.exists()
False
```

Using a text editor or some other means, create a blank text file called `hello.txt` in your home directory. Then rerun the above code example, making sure `path.exists()` returns `True`.

You can check whether a file path refers to a file or a directory. To check if the path is a file, use the `.is_file()` method:

```
>>> path.is_file()
True
```

Note that `.is_file()` returns `False` if the file doesn't exist.

Use the `.is_dir()` method to check if the file path refers to a directory:

```
>>> # "hello.txt" is not a directory
>>> path.is_dir()
False

>>> # home is a directory
>>> home.is_dir()
True
```

Working with file paths is an essential part of any programming project that reads or writes data from a hard drive or other storage

device. Understanding the differences between file paths on different operating systems and how to work with `pathlib.Path` objects so that your programs can work on any operating system is an important and useful skill.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a new `Path` object to a file called `my_file.txt` in a folder called `my_folder/` in your computer's home directory. Assign this `Path` object to the variable name `file_path`.
2. Check whether the path assigned to `file_path` exists.
3. Print the name of the path assigned to `file_path`. The output should be `my_file.txt`.
4. Print the name of the parent directory of the path assigned to `file_path`. The output should be `my_folder`.

12.3 Common File System Operations

Now that you know how to work with file paths using the `pathlib` module, let's take a look at some common file operations and how to perform them in Python.

Creating Directories and Files

To create a new directory, use the `Path.mkdir()` method. In IDLE's interactive window, type the following:

```
>>> from pathlib import Path  
>>> new_dir = Path.home() / "new_directory"  
>>> new_dir.mkdir()
```

After importing the `Path` class, you create a new path to a directory called `new_directory/` in your home folder and assign this path to the `new_dir` variable. Then you use `.mkdir()` to create the new directory.

You can now check that the new directory exists and is, in fact, a directory:

```
>>> new_dir.exists()  
True  
>>> new_dir.is_dir()  
True
```

If you try to create a directory that already exists, then you get an error:

```
>>> new_dir.mkdir()  
Traceback (most recent call last):  
  File "<pyshell#32>", line 1, in <module>  
    new_dir.mkdir()  
  File "C:\Users\David\AppData\Local\Programs\Python\  
Python\lib\pathlib.py", line 1266, in mkdir  
    self._accessor.mkdir(self, mode)  
FileExistsError: [WinError 183] Cannot create a file when  

```

When you call `.mkdir()`, Python attempts to create the `new_directory/` folder again. Since the directory already exists, this operation fails and a `FileExistsError` exception is raised.

What if you want to create a new directory only if it doesn't already exist, and you also want to avoid raising the `FileExistsError` if the directory *does* exist?

In that case, you can set the `exist_ok` parameter of `.mkdir()` to `True`:

```
>>> new_dir.mkdir(exist_ok=True)
```

When you execute `.mkdir()` with the `exist_ok` parameter set to `True`, the directory is created only if it doesn't already exist. If it does already exist, then nothing happens.

Setting `exist_ok` to `True` when calling `.mkdir()` is equivalent to the following code:

```
>>> if not new_dir.exists():
...     new_dir.mkdir()
```

Although the above code works just fine, setting the `exist_ok` parameter to `True` is shorter and doesn't sacrifice readability.

Now let's see what happens if you try to create a subdirectory within a directory that doesn't exist:

```
>>> nested_dir = new_dir / "folder_a" / "folder_b"
>>> nested_dir.mkdir()
Traceback (most recent call last):
  File "<pyshell#38>", line 1, in <module>
    nested_dir.mkdir()
  File "C:\Users\David\AppData\Local\Programs\Python\
    Python\lib\pathlib.py", line 1266, in mkdir
      self._accessor.mkdir(self, mode)
FileNotFoundError: [WinError 3] The system cannot find the path
specified: 'C:\\\\Users\\\\David\\\\new_directory\\\\folder_a\\\\folder_b'
```

The problem is that the parent directory `folder_a/` doesn't exist. Typically, to create a directory, all of the parent directories of the target directory (in this case `folder_b/`) in the path must already exist.

To create any parent directories needed to create the target directory, set the optional `parents` parameter of `.mkdir()` to `True`:

```
>>> nested_dir.mkdir(parents=True)
```

Now `.mkdir()` creates the parent directory, `folder_a/`, so that the target directory, `folder_b/`, can be created.

Putting all this together, you get the following common pattern for creating directories:

```
path.mkdir(parents=True, exist_ok=True)
```

When you set both the `parents` and `exist_ok` parameters to `True`, the entire path is created if needed, and no exception is raised if the path already exists.

This pattern is useful, but it may not be the right approach in every situation. For example, if a user inputs a nonexistent path, then you may wish to instead catch an exception so you can ask the user to verify the path that they entered. They might have just mistyped the name of an existing directory!

Now let's look at how to create files. Create a new `Path` object called `file_path` for the path `new_directory/file1.txt`:

```
>>> file_path = new_dir / "file1.txt"
```

There's no file called `file1.txt` in `new_directory/`, so the path doesn't exist yet:

```
>>> file_path.exists()  
False
```

You can create the file using the `Path.touch()` method:

```
>>> file_path.touch()
```

This creates a new file called `file1.txt` in the `new_directory/` folder. It doesn't contain any data yet, but the file exists:

```
>>> file_path.exists()  
True  
>>> file_path.is_file()  
True
```

Unlike `.mkdir()`, the `.touch()` method does not raise an exception if the path being created already exists:

```
>>> # Calling .touch() a second time doesn't raise an exception
>>> file_path.touch()
```

When you create a file using `.touch()`, the file doesn't contain any data. You'll learn how to write data to a file in section 12.5, "Reading and Writing Files."

You can't create a file in a directory that doesn't exist:

```
>>> file_path = new_dir / "folder_c" / "file2.txt"
>>> file_path.touch()
Traceback (most recent call last):
  File "<pyshell#47>", line 1, in <module>
    file_path.touch()
  File "C:\Users\David\AppData\Local\Programs\Python\
    Python\lib\pathlib.py", line 1256, in touch
    fd = self._raw_open(flags, mode)
  File "C:\Users\David\AppData\Local\Programs\Python\
    Python\lib\pathlib.py", line 1063, in _raw_open
    return self._accessor.open(self, flags, mode)
FileNotFoundError: [Errno 2] No such file or directory:
'C:\\\\Users\\\\David\\\\new_directory\\\\folder_c\\\\file2.txt'
```

The `FileNotFoundError` is raised because the `new_directory/` folder has no `folder_c/` subfolder.

Unlike `.mkdir()`, the `.touch()` method has no `parents` parameter that you can set to automatically create parent directories. This means that you need to first create any necessary directories before calling `.touch()` to create the file.

For instance, you can use `.parent` to get the path to the parent folder for `file2.txt` and then call `.mkdir()` to create the directory:

```
>>> file_path.parent.mkdir()
```

Since `.parent` returns a `Path` object, you can chain the `.mkdir()` method to write the entire operation on a single line of code.

With the `folder_c/` directory created, you can successfully create the file:

```
>>> file_path.touch()
```

Now that you know how to create files and directories, let's look at how to get the contents of a directory.

Iterating Over Directory Contents

Using `pathlib`, you can iterate over the contents of a directory. You might need to do this in order to process all the files in a directory.

The word *process* is vague. It could mean reading the file and extracting some data, compressing files in the directory, or some other operation.

For now, let's focus on how you go about retrieving the contents of a given directory. You'll learn how to read data from files in the next section.

Everything in a directory is either a file or a subdirectory. The `Path.iterdir()` method returns an iterator over `Path` objects representing each item in the directory.

To use `.iterdir()`, you first need a `Path` representing a directory. Let's use the `new_directory/` folder that you previously created in your home directory and assigned to the `new_dir` variable:

```
>>> for path in new_dir.iterdir():
...     print(path)
...
C:\Users\David\new_directory\file1.txt
C:\Users\David\new_directory\folder_a
C:\Users\David\new_directory\folder_c
```

Right now, this `new_directory/` folder contains three items:

1. A file called `file1.txt`
2. A directory called `folder_c/`
3. A directory called `folder_a/`

`.iterdir()` returns an iterable, so you can convert it to a list:

```
>>> list(new_dir.iterdir())
[WindowsPath('C:/Users/David/new_directory/file1.txt'),
 WindowsPath('C:/Users/David/new_directory/folder_a'),
 WindowsPath('C:/Users/David/new_directory/folder_c')]
```

You won't often need to convert this to a list. Generally, you'll use `.iterdir()` in a `for` loop like you did in the first example.

Notice that `.iterdir()` returns only those items that are directly contained in the `new_directory/` folder. That is, you can't see the path to the file that exists in the `folder_c/` directory.

There is a way to iterate over the contents a directory and all of its subdirectories, but you can't do it easily with `.iterdir()`. We'll get to this task in a moment, but first let's talk about how to search for files within a directory.

Searching for Files in a Directory

Sometimes you only need to iterate over files of a certain type or files with certain naming schemes. You can use the `Path.glob()` method

on a path representing a directory to get an iterable over directory contents that meet some criteria.

It might seem strange that a method that searches for files is called `.glob()`, but there's a historical reason behind this name. In early versions of the Unix operating system, a program called `glob` was used to expand file path patterns to full file paths.

The `.glob()` method does something similar. You pass to the method a string containing a pattern with a wildcard character, and `.glob()` returns a list of file paths that match the pattern.

A **wildcard character** is a special character that acts as a placeholder in a **pattern**. It's replaced by other characters to create a concrete file path. For example, in the pattern `"*.txt"`, the asterisk (*) is a wildcard character that can be replaced by any number of other characters.

The pattern `"*.txt"` matches any file path that ends with `.txt`. That is, if replacing the * in the pattern with every character in a file path up to the last four characters results in the original file path, then that file path is a **match** for the pattern `"*.txt"`.

Let's look at an example using the `new_directory/` folder previously assigned to the `new_dir` variable:

```
>>> for path in new_dir.glob("*.txt"):
...     print(path)
...
C:\Users\David\new_directory\file1.txt
```

Like `.iterdir()`, the `.glob()` method returns an iterable of paths, but this time only those paths that match the pattern `"*.txt"` are returned. Note that `.glob()` returns only the paths that are directly contained in the folder on which it is called.

You can convert the return value of `.glob()` to a list:

```
>>> list(new_dir.glob("*.*txt"))
[WindowsPath('C:/Users/David/new_directory/file1.txt')]
```

You will most often use `.glob()` in a `for` loop.

The following table describes some common wildcard characters:

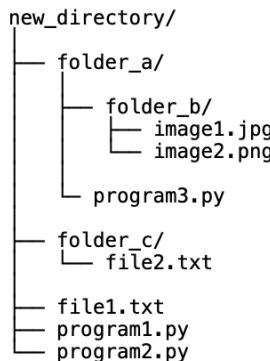
Wildcard Character	Description	Example	Matches	Does Not Match
*	Any number of characters	"*b*"	b, ab, bc, abc	a, c, ac
?	A single character	"?bc"	abc, bbc, cbc	bc, aabc, abcd
[abc]	Matches one character in the brackets	[CB]at	Cat, Bat	at, cat, bat

We'll look at some examples of each of the wildcard characters later. But first, let's create a few more files in the `new_directory/` folder so that we have more options to play with.

Type in the following code:

```
>>> paths = [
...     new_dir / "program1.py",
...     new_dir / "program2.py",
...     new_dir / "folder_a" / "program3.py",
...     new_dir / "folder_a" / "folder_b" / "image1.jpg",
...     new_dir / "folder_a" / "folder_b" / "image2.png",
... ]
>>> for path in paths:
...     path.touch()
...
>>>
```

After you execute the above, the `new_directory/` folder has the following structure:



Now that we have a more interesting structure to work with, let's see how `.glob()` works with each of the wildcard characters.

The * Wildcard

The `*` wildcard matches any number of characters in a file path pattern. For example, the pattern `"*.py"` matches all file paths that end in `.py`:

```
>>> list(new_dir.glob("*.py"))
[WindowsPath('C:/Users/David/new_directory/program1.py'),
 WindowsPath('C:/Users/David/new_directory/program2.py')]
```

You can use the `*` wildcard multiple times in a single pattern:

```
>>> list(new_dir.glob("*1*"))
[WindowsPath('C:/Users/David/new_directory/file1.txt'),
 WindowsPath('C:/Users/David/new_directory/program1.py')]
```

The pattern `"*1*"` matches any file path containing the number `1` with any number of characters before and after it. The only files in the `new_directory/` folder that contain the number `1` are `file1.txt` and `program1.py`.

If you leave off the first * from the pattern "*1*" to get the pattern "1*", then nothing gets matched:

```
>>> list(new_dir.glob("1*"))
[]
```

The pattern "1*" matches files paths that start with the number 1 and are followed by any number of characters after it. There are no files in the new_directory/ folder that match this pattern, so .glob() doesn't return anything.

The ? Wildcard

The ? wildcard character matches a single character in a pattern. For example, the pattern "program?.py" will match any file path that starts with the word program and is followed by a single character and then .py:

```
>>> list(new_dir.glob("program?.py"))
[WindowsPath('C:/Users/David/new_directory/program1.py'),
 WindowsPath('C:/Users/David/new_directory/program2.py')]
```

You can use multiple instances of ? in a single pattern:

```
>>> list(new_dir.glob("?older_?"))
[WindowsPath('C:/Users/David/new_directory/folder_a'),
 WindowsPath('C:/Users/David/new_directory/folder_c')]
```

The pattern "?older_?" matches paths that start with any letter followed by older_ and some other character. In the new_directory/ folder, those paths are the folder_a/ and folder_c/ directories.

You can also combine the * and ? wildcards:

```
>>> list(new_dir.glob("*1.??"))
[WindowsPath('C:/Users/David/new_directory/program1.py')]
```

The pattern "*1.???" matches any file path that contains a 1 followed by a dot (.) and two more characters. The only path in new_directory/

matching this pattern is `program1.py`. Notice that `file1.txt` doesn't match the pattern because the dot is followed by three characters.

The [] Wildcard

The `[]` wildcard works kind of like the `?` wildcard because it matches only a single character. The difference is that, instead of matching any single character like `?` does, `[]` matches only those characters that are between the square brackets.

For example, the pattern `"program[13].py"` matches any path containing the word `program`, followed by either a `1` or a `3` and the extension `.py`. In the `new_directory/` folder, `program1.py` is the only path matching this pattern:

```
>>> list(new_dir.glob("program[13].py"))
[WindowsPath('C:/Users/David/new_directory/program1.py')]
```

As with the other wildcards, you can use multiple instances of the `[]` wildcard as well as combine it with any of the others.

Recursive Matching With the `**` Wildcard

The major limitation you've seen with both `.iterdir()` and `.glob()` is that they return only those paths that are directly contained in the folder on which they're called.

For example, `new_dir.glob("*.txt")` returns only the `file1.txt` path in `new_directory/`. It doesn't return the `file2.txt` path in the `folder_c/` subdirectory even though that path matches the `"*.txt"` pattern.

There's a special wildcard character `**` that makes the pattern recursive. The common way to use it is to prefix your pattern with `"**/"`. This tells `.glob()` to match your pattern both in the current directory and in any of its subdirectories.

For example, the pattern "`**/*.txt`" matches both `file1.txt` and `folder_c/file2.txt`:

```
>>> list(new_dir.glob("**/*.txt"))
[WindowsPath('C:/Users/David/new_directory/file1.txt'),
 WindowsPath('C:/Users/David/new_directory/folder_c/file2.txt')]
```

Similarly, the pattern "`**/*.py`" matches any `.py` files in `new_directory` and in any of its subdirectories:

```
>>> list(new_dir.glob("**/*.py"))
[WindowsPath('C:/Users/David/new_directory/program1.py'),
 WindowsPath('C:/Users/David/new_directory/program2.py'),
 WindowsPath('C:/Users/David/new_directory/folder_a/program3.py')]
```

There's also a shorthand method of recursive matching called `.rglob()`. To use it, pass the pattern without the "`**/`" prefix:

```
>>> list(new_dir.rglob("*.py"))
[WindowsPath('C:/Users/David/new_directory/program1.py'),
 WindowsPath('C:/Users/David/new_directory/program2.py'),
 WindowsPath('C:/Users/David/new_directory/folder_a/program3.py')]
```

The `r` in `.rglob()` stands for recursive. Some people prefer to use this method instead of prefixing their patterns with "`**/`" because it's slightly shorter. Both versions are perfectly valid.

In this book, we'll use `.rglob()` instead of the `**/` prefix.

Moving and Deleting Files and Folders

Sometimes you need to move a file or directory to a new location or delete a file or directory altogether. You can do this using `pathlib`, but keep in mind that doing so can result in the loss of data, so these operations must be made with extreme care.

To move a file or directory, use the `.replace()` method. For example, the following moves the `file1.txt` file in the `new_directory/` folder to the `folder_a/` subfolder:

```
>>> source = new_dir / "file1.txt"
>>> destination = new_dir / "folder_a" / "file1.txt"
>>> source.replace(destination)
WindowsPath('C:/Users/David/new_directory/folder_a/file1.txt')
```

Here, you call `.replace()` on the source path. You pass the destination path to `.replace()` as a single argument. Notice that `.replace()` returns the path to the new location of the file.

Important

If the destination path already exists, then `.replace()` overwrites the destination with the source file without raising any kind of exception. This can cause undesired loss of data if you aren't careful.

You may want to first check if the destination file exists and move the file only if it doesn't:

```
if not destination.exists():
    source.replace(destination)
```

You can also use `.replace()` to move or rename an entire directory. For instance, the following code renames the `folder_c` subdirectory of `new_directory/` to `folder_d/`:

```
>>> source = new_dir / "folder_c"
>>> destination = new_dir / "folder_d"
>>> source.replace(destination)
WindowsPath('C:/Users/David/new_directory/folder_d')
```

Again, if the destination folder already exists, then it's completely replaced by the source folder, which could result in the loss of quite a bit of data.

To delete a file, use the `.unlink()` method:

```
>>> file_path = new_dir / "program1.py"
>>> file_path.unlink()
```

This deletes the `program1.py` file in the `new_directory/` folder, which you can check with `.exists()`:

```
>>> file_path.exists()
False
```

You can also see that it's been removed using `.iterdir()`:

```
>>> list(new_dir.iterdir())
[WindowsPath('C:/Users/David/new_directory/folder_a'),
 WindowsPath('C:/Users/David/new_directory/folder_d'),
 WindowsPath('C:/Users/David/new_directory/program2.py')]
```

If the path that you call `.unlink()` on doesn't exist, then a `FileNotFoundException` exception is raised:

```
>>> file_path.unlink()
Traceback (most recent call last):
  File "<pyshell#94>", line 1, in <module>
    file_path.unlink()
  File "C:\Users\David\AppData\Local\Programs\Python\
    Python\lib\pathlib.py", line 1303, in unlink
    self._accessor.unlink(self)
FileNotFoundException: [WinError 2] The system cannot find the file
specified: 'C:\\\\Users\\\\David\\\\new_directory\\\\program1.py'
```

If you want to ignore the exception, then set the optional `missing_ok` parameter to `True`:

```
>>> file_path.unlink(missing_ok=True)
```

In this case, nothing happens because the file located at `file_path` doesn't exist.

Important

When you delete a file, it's gone forever. Make sure you really want to delete it before you proceed!

You can use `.unlink()` only with paths representing files. To remove a directory instead, you can use the `.rmdir()` method. Keep in mind that the folder must be empty. Otherwise, the operation will raise an `OSError` exception:

```
>>> folder_d = new_dir / "folder_d"  
>>> folder_d.rmdir()  
Traceback (most recent call last):  
  File "<pyshell#97>", line 1, in <module>  
    folder_d.rmdir()  
  File "C:\Users\David\AppData\Local\Programs\Python\  
    Python\lib\pathlib.py", line 1314, in rmdir  
      self._accessor.rmdir(self)  
 OSError: [WinError 145] The directory is not empty:  

```

In the case of `folder_d/`, it contains only a single file called `file2.txt`. To delete `folder_d/`, first delete all of the files it contains:

```
>>> for path in folder_d.iterdir():  
...     path.unlink()  
...  
>>> folder_d.rmdir()
```

Now `folder_d/` is deleted:

```
>>> folder_d.exists()  
False
```

If you need to delete an entire directory even if it's non-empty, then `pathlib` won't help you much. However, the built-in `shutil` module includes a `rmtree()` function that you can use to delete directories populated with files.

Here's how you use `rmtree()` to delete `folder_a/`:

```
>>> import shutil  
>>> folder_a = new_dir / "folder_a"  
>>> shutil.rmtree(folder_a)
```

Recall that `folder_a/` contains a subfolder, `folder_b/`, which itself contains two files called `image1.jpg` and `image2.png`.

When you pass the `folder_a` path object to `rmtree()`, `folder_a/` and all of its contents are deleted:

```
>>> # The folder_a/ directory no longer exists  
>>> folder_a.exists()  
False  
>>> # Searching for `image*.*` files returns nothing  
>>> list(new_dir.rglob("image*.*"))  
[]
```

In this section, you covered quite a bit of ground. You learned how to perform several common file system operations:

- Creating files and directories
- Iterating over the contents of a directory
- Searching for files and folders using wildcards
- Moving and deleting files and folders

All of these are common tasks. It's extremely important, however, to remember that your programs are guests on another person's computer. If you aren't careful, you can inadvertently cause damage to a user's computer, resulting in the loss of important documents and other data.

You should always use caution when working with the file system. When in doubt, check that a file path exists before performing an operation, and always confirm with the user that what you're about to do is okay!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a new directory in your home folder called `my_folder/`.
2. Inside `my_folder/` create three files:
 - `file1.txt`
 - `file2.txt`
 - `image1.png`
3. Move the file `image1.png` to a new directory called `images/` inside the `my_folder/` directory.
4. Delete the file `file1.txt`.
5. Delete the `my_folder/` directory.

12.4 Challenge: Move All Image Files to a New Directory

In the chapter 12 `practice_files` folder, there is a subfolder called `documents/`. The directory contains several files and subfolders. Some of the files are images ending with the `.png`, `.gif`, or `.jpg` file extension.

Create a new folder in the `practice_files` folder called `images/`, and move all image files to that folder. When you're done, the new folder should have four files in it:

1. `image1.png`
2. `image2.gif`
3. `image3.png`
4. `image4.jpg`

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

12.5 Reading and Writing Files

Files are abundant in the modern world. They're the medium in which data is digitally stored and transferred. Chances are, you've opened dozens, if not hundreds, of files just today.

In this section, you'll learn how to read and write files with Python.

What Is a File?

A **file** is a sequence of bytes and a **byte** is a number between 0 and 255. That is, a file is a sequence of integer values.

The bytes in a file must be **decoded** into something meaningful in order to understand the contents of the file.

Python has standard library modules for working with text, CSV, and audio files. There are a number of third-party packages available for working with other file types.

You'll learn how to install third-party packages in chapter 13, "Installing Packages with pip." You'll also see how to work with PDF files in chapter 14, "Creating and Modifying PDF Files."

In this section, you'll learn how to work with plain text files.

Understanding Text Files

Text files are files that contain only text. They're perhaps the easiest files to work with. There are two issues, though, that can be frustrating when working with text files:

1. Character encoding
2. Line endings

Before jumping into reading and writing text files, let's look at what these issues are so that you know how to deal with them effectively.

Character Encoding

Text files are stored on disk as a sequence of bytes. Each byte, or group of bytes in some cases, represents a different character in the file.

When text files are written, characters typed on the keyboard are converted into bytes in a process called **encoding**. When a text file is read, the bytes are **decoded** back into text.

The integer a character is associated with is determined by the file's **character encoding**. There are many character encodings. Here are four of the most widely used character encodings:

1. ASCII
2. UTF-8
3. UTF-16
4. UTF-32

Some character encodings, such as ASCII and UTF-8, encode characters the same way. For example, numbers and English letters are encoded the same way in both ASCII and UTF-8.

The difference between ASCII and UTF-8 is that UTF-8 can encode more characters. ASCII can't encode characters like ñ or ü, but UTF-8 can. This means that you can decode ASCII-encoded text with UTF-8, but you can't always decode UTF-8-encoded text with ASCII.

Important

Serious problems may occur when different encodings are used to encode and decode text.

For instance, text encoded as UTF-8 that is decoded with UTF-16 may be interpreted as an entirely different language than originally intended!

For a thorough introduction to character encodings, check out [Real Python's “Unicode & Character Encodings in Python: A Painless Guide.”](#)

Knowing which encoding a file uses is important, but it isn't always obvious. On modern Windows computers, text files are usually encoded with UTF-16 or UTF-8. On macOS and Ubuntu Linux, the default character encoding is usually UTF-8.

For the remainder of this section, we'll assume that the character encoding of all text files that we work with is UTF-8. If you encounter problems, then you may need to alter the examples to use a different encoding.

Line Endings

Each line in a text file ends with one or two characters that indicate the line has ended. These characters aren't usually displayed in a text editor, but they exist as bytes in the file data.

The two characters used to represent line endings are the **carriage return** and **line feed** characters. In Python strings, these characters are represented by the escape sequences `\r` and `\n`, respectively.

On Windows, line endings are represented by default with both a carriage return and a line feed. On macOS and most Linux distributions, line endings are represented with just a single line feed character.

When you read a Windows file on macOS or Linux, you'll sometimes see extra blank lines between lines of text. This is because the carriage

return also represents a line ending on macOS and Linux.

For example, suppose the following text file was created in Windows:

```
Pug\r\n
Jack Russell Terrier\r\n
English Springer Spaniel\r\n
German Shepherd\r\n
```

On macOS or Ubuntu, this file is interpreted with double spacing between lines:

```
Pug\r
\n
Jack Russell Terrier\r
\n
English Springer Spaniel\r
\n
German Shepherd\r
\n
```

In practice, the differences between line endings on different operating systems is rarely problematic. Python can handle line ending conversions for you automatically, so you don't have to worry about it too often.

Python File Objects

Files are represented in Python with **file objects**, which are instances of classes designed to work with different types of files.

Python has a couple of different types of file objects:

1. **Text file objects** are used for interacting with text files.
2. **Binary file objects** are used for working directly with the bytes contained in files.

Text file objects handle encoding and decoding bytes for you. All you need to do is specify which character encoding to use. Binary file objects, on the other hand, do not perform any kind of encoding or decoding.

There are two ways to create a file object in Python:

1. The `Path.open()` method
2. The built-in `open()` function

Let's look at each of these.

The `Path.open()` Method

To use the `Path.open()` method, you first need a `Path` object. In IDLE's interactive window, execute the following:

```
>>> from pathlib import Path  
>>> path = Path.home() / "hello.txt"  
>>> path.touch()  
>>> file = path.open(mode="r", encoding="utf-8")
```

First, you create a `Path` object for the `hello.txt` file and assign it to the `path` variable. Then `path.touch()` creates the file in your home directory. Finally, `.open()` returns a new file object representing the `hello.txt` file and assigns it to the `file` variable.

Two keyword parameters are used to open the file:

1. The `mode` parameter determines in which mode the file should be opened. The "`r`" argument opens the file in **read mode**.
2. The `encoding` parameter determines the character encoding used to decode the file. The argument "`utf-8`" represents UTF-8 character encoding.

You can inspect the `file` variable to see that it's assigned to a text file object:

```
>>> file
<_io.TextIOWrapper name='C:\Users\David\hello.txt' mode='r'
encoding='utf-8'>
```

Text file objects are instances of the `TextIOWrapper` class. You'll never need to instantiate this class directly since you can create it with the `Path.open()` method.

There are a number of different modes you can use to open a file. These are described in the following table:

Mode	Description
"r"	Creates a text file object for reading and raises an error if the file can't be opened
"w"	Creates a text file object for writing and overwrites all existing data in the file
"a"	Creates a text file object for appending data to the end of a file
"rb"	Creates a binary file object for reading and raises an error if the file can't be opened
"wb"	Creates a binary file object for writing and overwrites all existing data in the file
"ab"	Creates a binary file object for appending data to the end of the file

The strings for some of the most commonly used character encodings can be found in the table below:

String	Character Encoding
"ascii"	ASCII
"utf-8"	UTF-8
"utf-16"	UTF-16
"utf-32"	UTF-32

When you create a file object with `.open()`, Python maintains a link to the file resource until you either explicitly tell Python to close the file or the program ends.

Important

You should always explicitly tell Python to close a file.

Forgetting to close opened files is like littering. When your program stops running, it shouldn't leave unnecessary waste lying around the system.

To close a file, use the file object's `.close()` method:

```
>>> file.close()
```

Using `Path.open()` is the preferred way to open a file when you have an existing `Path` object, but there's also a built-in function called `open()` that you can use to open a file.

The Built-in `open()` Function

The built-in `open()` function works almost exactly like the `Path.open()` method, except that its first parameter is a string containing the path to the file you want to open.

First, create a new variable called `file_path` and assign to it a string containing the path to the `hello.txt` file you created above:

```
>>> file_path = "C:/Users/David/hello.txt"
```

Note that you'll need to change the path to match the path of the file on your own computer.

Next, create a new file object using the built-in `open()` and assign it to the variable `file`:

```
>>> file = open(file_path, mode="r", encoding="utf-8")
```

The first parameter of `open()` must be a path string. The `mode` and `encoding` parameters are the same as the parameters for the `Path.open()` method. In this example, `mode` is set to "r" for read mode, and `encoding` is set to "utf-8".

Just like the file object returned by `Path.open()`, the file object returned by `open()` is a `TextIOWrapper` instance:

```
>>> file
<_io.TextIOWrapper name='C:/Users/David/hello.txt' mode='r'
encoding='utf-8'>
```

To close the file, use the file object's `.close()` method:

```
>>> file.close()
```

For the most part, you'll use the `Path.open()` method to open a file from an existing `pathlib.Path` object. However, if you don't need all of the functionality of the `pathlib` module, then `open()` is a great way to quickly create a file object.

The `with` Statement

When you open a file, your program accesses data external to the program itself. The operating system must manage the connection between your program and the physical file. When you call a file object's `.close()` method, the operating system knows to close the connection.

If your program crashes between the time a file is opened and when it's closed, then the system resources used by the connection may live on until the operating system realizes they're no longer needed.

To ensure that file system resources are cleaned up even if a program crashes, you can open a file in a `with` statement. The pattern for using the `with` statement looks like this:

```
with path.open(mode="r", encoding="utf-8") as file:
    # Do something with file
```

The `with` statement has two parts: a header and a body. The header always starts with the `with` keyword and ends with a colon (:). The return value of `path.open()` is assigned to the variable name after the `as` keyword.

After the `with` statement header is an indented block of code. When code execution leaves the indented block, the file object assigned to `file` is closed automatically, even if an exception is raised during execution of the code inside of the block.

`with` statements also work with the built-in `open()`:

```
with open(file_path, mode="r", encoding="utf-8") as file:  
    # Do something with file
```

There really is no reason *not* to open files in a `with` statement. It's considered the Pythonic way of working with files. For the rest of this book, we'll use this pattern whenever we open a file.

Reading Data From a File

Using a text editor, open the `hello.txt` file in your home directory that you previously created and type the text `Hello, World` into it. Then save the file.

In IDLE's interactive window, type the following:

```
>>> path = Path.home() / "hello.txt"  
>>> with path.open(mode="r", encoding="utf-8") as file:  
...     text = file.read()  
...  
>>>
```

The file object created by `path.open()` is assigned to the `file` variable. Inside of the `with` block, the file object's `.read()` method reads the text from the file and assigns the result to the variable `text`.

The value returned by `.read()` is a string object with the value "Hello, World":

```
>>> type(text)
<class 'str'>
>>> text
'Hello, World'
```

The `.read()` method reads the text in the file and returns it as a string.

If there are multiple lines of text in the file, then each line in the string is separated with a newline character (`\n`). In a text editor, open the `hello.txt` file again and put the text "Hello again" on the second line. Then save the file.

Back in IDLE's interactive window, read the text from the file again:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
...     text = file.read()
...
>>> text
'Hello, World\nHello again'
```

The text from each line has a `\n` character in between.

Instead of reading the entire file at once, you can process each line of the file one at a time:

```
>>> with path.open(mode="r", encoding="utf-8") as file:
...     for line in file.readlines():
...         print(line)
...
Hello, World

Hello again
```

The `.readlines()` method returns an iterable of lines from the file. At each step of the `for` loop, the next line of text in the file is returned and printed.

Notice the extra blank line between the two lines of text. This isn't caused by line endings in the file. It happens because `print()` automatically inserts a newline character at the end of every string it prints.

To print the two lines without the extra blank line, set the `print()` function's optional `end` parameter to an empty string:

```
>>> with path.open(mode="r", encoding="utf-8") as file:  
...     for line in file.readlines():  
...         print(line, end="")  
...  
Hello, World  
Hello again
```

You'll often want to use `.readlines()` instead of `.read()`. For example, each line in a file might represent a single record. You can use `.readlines()` to loop over the lines and process them as needed.

If you try to read from a file that doesn't exist, then both `Path.open()` and the built-in `open()` raise a `FileNotFoundException`:

```
>>> path = Path.home() / "new_file.txt"  
>>> with path.open(mode="r", encoding="utf-8") as file:  
...     text = file.read()  
...  
Traceback (most recent call last):  
  File "<pyshell#197>", line 1, in <module>  
    with path.open(mode="r", encoding="utf-8") as file:  
  File "C:\Users\David\AppData\Local\Programs\Python\  
Python\lib\pathlib.py", line 1200, in open  
    return io.open(self, mode, buffering, encoding, errors, newline,  
  File "C:\Users\David\AppData\Local\Programs\Python\  
Python\lib\pathlib.py", line 1054, in _opener  
    return self._accessor.open(self, flags, mode)  
FileNotFoundException: [Errno 2] No such file or directory:  
'C:\\\\Users\\\\David\\\\new_file.txt'
```

Next, let's see how to write data to a file.

Writing Data to a File

To write data to a plain text file, you pass a string to a file object's `.write()` method. The file object must be opened in **write mode** by passing the value "w" to the `mode` parameter.

For instance, the following writes the text "Hi there!" to the `hello.txt` file in your home directory:

```
>>> with path.open(mode="w", encoding="utf-8") as file:  
...     file.write("Hi there!")  
...  
9  
>>>
```

Notice that the integer 9 is displayed after the `with` block executes. That's because `.write()` returns the number of characters it writes. The string "Hi there!" has nine characters, so `.write()` returns 9.

When the text "Hi there!" is written to the `hello.txt` file, any existing contents are overwritten. It's as if you deleted the old `hello.txt` file and created a new one.

Important

When you set `mode="w"` in `.open()`, the contents of the original file are overwritten. This results in the loss of all of the original data in the file!

You can verify that the file contains only the text "Hi there!" by reading and displaying the contents of the file:

```
>>> with path.open(mode="r", encoding="utf-8") as file:  
...     text = file.read()  
...  
>>> print(text)  
Hi there!
```

You can **append** data to the end of a file by opening the file in **append mode**:

```
>>> with path.open(mode="a", encoding="utf-8") as file:  
...     file.write("\nHello")  
...  
6
```

When a file is opened in append mode, new data is written to the end of the file and old data is left intact. The newline character is put at the beginning of the string so that the word "Hello" is printed on a new line at the end of the file.

Without a newline character at the beginning of the string, the word "Hello" would be printed on the same line as any existing text at the end of the file.

You can check that the word "Hello" is written to the second line by opening and reading from the file:

```
>>> with path.open(mode="r", encoding="utf-8") as file:  
...     text = file.read()  
...  
>>> print(text)  
Hi there!  
Hello
```

You can write multiple lines to a file at the same time using the `.writelines()` method. First, create a list of strings:

```
>>> lines_of_text = [  
...     "Hello from Line 1\n",  
...     "Hello from Line 2\n",  
...     "Hello from Line 3 \n",  
... ]
```

Then open the file in write mode and use `.writelines()` to write each string in the list to the file:

```
>>> with path.open(mode="w", encoding="utf-8") as file:  
...     file.writelines(lines_of_text)  
...  
>>>
```

Each string in `lines_of_text` is written to the file. Notice that each string ends with the newline character (`\n`). That's because `.writelines()` doesn't automatically write each string on a new line.

If you open a nonexistent path in write mode, then Python will create the file as long as all the parent folders in the path exist:

```
>>> path = Path.home() / "new_file.txt"  
>>> with path.open(mode="w", encoding="utf-8") as file:  
...     file.write("Hello!")  
...  
6
```

Since the `Path.home()` directory exists, a new file called `new_file.txt` is created automatically. However, if one of the parent directories does *not* exist, then `.open()` will raise a `FileNotFoundException`:

```
>>> path = Path.home() / "new_folder" / "new_file.txt"  
>>> with path.open(mode="w", encoding="utf-8") as file:  
...     file.write("Hello!")  
...  
Traceback (most recent call last):  
  File "<pyshell#172>", line 1, in <module>  
    with path.open(mode="w", encoding="utf-8") as file:  
  File "C:\Users\David\AppData\Local\Programs\Python\  
    Python\lib\pathlib.py", line 1200, in open  
        return io.open(self, mode, buffering, encoding, errors, newline,  
  File "C:\Users\David\AppData\Local\Programs\Python\  
    Python\lib\pathlib.py", line 1054, in _opener  
        return self._accessor.open(self, flags, mode)  
FileNotFoundException: [Errno 2] No such file or directory:  
'C:\\\\Users\\\\David\\\\new_folder\\\\new_file.txt'
```

If you want to write to a path with parent folders that may not exist, then call the `.mkdir()` method with the `parents` parameter set to `True` before opening the file in write mode:

```
>>> path.parent.mkdir(parents=True)
>>> with path.open(mode="w", encoding="utf-8") as file:
...     file.write("Hello!")
...
6
```

In this section, you covered a lot of ground. You learned that all files are sequences of bytes, which are integers with values between 0 and 255.

You also learned about character encodings, which are used to translate between bytes and text, and the differences between line endings on different operating systems.

Finally, you saw how to read and write text files using the `Path.open()` method and the built-in `open()` function.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write the following text to a file in your home directory called `starships.txt`:

```
Discovery
Enterprise
Defiant
Voyager
```

Each word should be on a separate line.

2. Read the file `starhips.txt` that you created in exercise 1 and print each line of text in the file. The output should not have extra blank lines between each word.

3. Read the file `startships.txt` and print the names of the starships that start with the letter D.

12.6 Read and Write CSV Data

Suppose you have a temperature sensor in your house that records the temperature every four hours. Over the course of a day, it takes six temperature readings.

You can store each temperature reading in a list:

```
>>> temperature_readings = [68, 65, 68, 70, 74, 72]
```

Each day the sensor generates a new list of numbers. To store these values to a file, you can write the values from each day on a new line in a text file and separate each value with a comma:

```
>>> from pathlib import Path
>>> file_path = Path.home() / "temperatures.csv"
>>> with file_path.open(mode="a", encoding="utf-8") as file:
...     file.write(str(temperature_readings[0]))
...     for temp in temperature_readings[1:]:
...         file.write(f", {temp}")
...
2
3
3
3
3
3
```

This creates a file in your home directory called `temperatures.csv` and opens it in append mode. On a new line at the end of the file, the first value in the `temperature_readings` list is written to the file. Then each remaining value in the list is written, preceded by a comma, to the same line.

The final string of text written to the file is "68,65,68,70,74,72". You can verify this by reading the text in the file:

```
>>> with file_path.open(mode="r", encoding="utf-8") as file:  
...     text = file.read()  
...  
>>> text  
'68,65,68,70,74,72'
```

This format is called **comma-separated values**, or **CSV**. The `temperatures.csv` file is called a **CSV file**.

CSV files are a great way to store records of sequential data because you can recover each row of the CSV value as a list:

```
>>> temperatures = text.split(",")  
>>> temperatures  
['68', '65', '68', '70', '74', '72']
```

In section 9.2, “Lists Are Mutable Sequences,” you learned how to create a list from a string using the `.split()` string method. In the example above, a new list is created from the `text` read from the `temperatures.csv` file.

The values in the `temperatures` list are strings, not integers like the values originally written to the file. This is because values read from a text file are always read as strings.

You can convert the strings to integers using a list comprehension:

```
>>> int_temperatures = [int(temp) for temp in temperatures]  
>>> int_temperatures  
[68, 65, 68, 70, 74, 72]
```

You’ve now recovered the list that you originally wrote to the `temperatures.csv` file!

What these examples illustrate is that a CSV file is a plain text file. Using techniques from section 12.5, “Reading and Writing Files,” you

can store sequences of values in the rows of the CSV file and then read from the file to recover the data.

Reading and writing CSV files is so common that the Python standard library has a module called `csv` to lessen the workload required for working with CSV files. In the following sections, you'll learn how to use the `csv` module to write to and read from CSV files.

The `csv` Module

The `csv` module can be used to read and write CSV files. In this section, you'll rework the previous example using the `csv` module so you can see how the module works and what operations it handles for you.

To get started, import the `csv` module in IDLE's interactive window:

```
>>> import csv
```

Let's start by creating a new CSV file containing several days' worth of temperature data.

Writing CSV Files With `csv.writer`

Create a list of lists containing three days' worth of temperature readings:

```
>>> daily_temperatures = [
...     [68, 65, 68, 70, 74, 72],
...     [67, 67, 70, 72, 72, 70],
...     [68, 70, 74, 76, 74, 73],
... ]
```

Now open the `temperatures.csv` file in write mode:

```
>>> file_path = Path.home() / "temperatures.csv"
>>> file = file_path.open(mode="w", encoding="utf-8", newline="")
```

Instead of using a `with` statement, you create a file object and assign it to the `file` variable so you can inspect each step of the writing process.

Important

In the above example, notice that the `newline` parameter in `.open()` is set to `""`.

This is because the `csv` module does its own newline conversions. If you don't specify `newline=""` when opening the file, then some systems, such as Windows, will interpret newlines incorrectly and insert a second newline after each line in the file.

Now create a new CSV writer object by passing the file object `file` to `csv.writer()`:

```
>>> writer = csv.writer(file)
```

`csv.writer()` returns a CSV writer object with methods for writing data to the CSV file.

For instance, you can use the `writer.writerow()` method to write a list to a new row in the CSV file:

```
>>> for temp_list in daily_temperatures:  
...     writer.writerow(temp_list)  
...  
19  
19  
19
```

Just like a file object's `.write()` method, `.writerow()` returns the number of characters written to the file. Each list in `daily_temperatures` gets converted to a string containing the temperatures separated by commas, and each of these strings has 19 characters.

Now close the file:

```
>>> file.close()
```

If you open the `temperatures.csv` file in a text editor, then you'll see the following text in the file:

```
68,65,68,70,74,72  
67,67,70,72,72,70  
68,70,74,76,74,73
```

In the above examples, you didn't use a `with` statement to write to the file so that you could inspect each operation in IDLE's interactive window. In practice, it's much better to use `with`.

Here's what the code looks like using the `with` statement:

```
with file_path.open(mode="w", encoding="utf-8", newline="") as file:  
    writer = csv.writer(file)  
    for temp_list in daily_temperatures:  
        writer.writerow(temp_list)
```

The main advantage of using `csv.writer` to write to a CSV file is that you don't need to worry about converting values to strings before writing them to the file. The `csv.writer` object handles this for you, which results in shorter and cleaner code.

The `.writerow()` method writes a single row to the CSV file, but you can write multiple rows at once using `.writerows()`. This shortens the code even more when your data is already in a list of lists:

```
with file_path.open(mode="w", encoding="utf-8", newline="") as file:  
    writer = csv.writer(file)  
    writer.writerows(daily_temperatures)
```

Now let's read from `temperatures.csv` to recover the `daily_temperatures` list of lists that you used to create the file.

Reading CSV Files With `csv.reader`

To read a CSV file with the `csv` module, you use the `csv.reader` class.

Like `csv.writer` objects, `csv.reader` objects are instantiated from a file object:

```
>>> file = file_path.open(mode="r", encoding="utf-8", newline="")
>>> reader = csv.reader(file)
```

`csv.reader()` returns a CSV reader object that can be used to iterate over the rows of the CSV file:

```
>>> for row in reader:
...     print(row)
...
[['68', '65', '68', '70', '74', '72'],
 ['67', '67', '70', '72', '72', '70'],
 ['68', '70', '74', '76', '74', '73']]
>>> file.close()
```

Each row of the CSV file is returned as a list of strings. To recover the `daily_temperatures` list of lists, you'll need to convert each list of strings to a list of integers using a list comprehension.

Here's a full example that opens the CSV file in a `with` statement, reads each row in the CSV file, converts the list of strings to a list of integers, and stores each list of integers in a list of lists called `daily_temperatures`:

```
>>> # Create an empty list
>>> daily_temperatures = []
>>> with file_path.open(mode="r", encoding="utf-8", newline="") as file:
...     reader = csv.reader(file)
...     for row in reader:
...         # Convert row to list of integers
...         int_row = [int(value) for value in row]
...         # Append the list of integers to daily_temperatures list
...         daily_temperatures.append(int_row)
...
>>> daily_temperatures
[[68, 65, 68, 70, 74, 72], [67, 67, 70, 72, 72, 70],
 [68, 70, 74, 76, 74, 73]]
```

It's much easier to work with CSV files using the `csv` module than it is to use the standard tools for reading and writing plain text files.

Sometimes, though, CSV files are more complex than a file with rows of values that all have the same type. Each row may represent a record with various fields, and the first row in the file may be a **header row** with the names of the fields.

Reading and Writing CSV Files With Headers

Here's an example of a CSV file with a header row containing multiple data types:

```
name,department,salary
Lee,Operations,75000.00
Jane,Engineering,85000.00
Diego,Sales,80000.00
```

The first line of the file contains field names. Each following line contains a record with a value for each field.

It's possible to read CSV files such as the one above using `csv.reader()`, but you have to keep track of the header row, and each row is returned as a list without the field names attached to it. It makes more sense to return each row as a dictionary whose keys are the field names and whose values are the field values in the row. This is precisely what `csv.DictReader` objects do!

Using a text editor, create a new CSV file in your home directory called `employees.csv` and save the example CSV text from above to it.

In IDLE's interactive window, open the `employees.csv` file and create a new `csv.DictReader` object:

```
>>> file_path = Path.home() / "employees.csv"
>>> file = file_path.open(mode="r", encoding="utf-8", newline="")
>>> reader = csv.DictReader(file)
```

When you create a `DictReader` object, the first row of the CSV file is

assumed to contain the field names. These values get stored in a list and are assigned to the `DictReader` instance's `.fieldnames` attribute:

```
>>> reader.fieldnames
['name', 'department', 'salary']
```

Just like `csv.reader` objects, `DictReader` objects are iterable:

```
>>> for row in reader:
...     print(row)
...
{'name': 'Lee', 'department': 'Operations', 'salary': '75000.000'}
{'name': 'Jane', 'department': 'Engineering', 'salary': '85000.00'}
{'name': 'Diego', 'department': 'Sales', 'salary': '80000.00'}
>>> file.close()
```

Instead of returning each row as a list, `DictReader` objects return each row as a dictionary. The dictionary's keys are the field names, and the values are the field values from each row in the CSV file.

Notice that the `salary` field is read as a string. Since CSV files are plain text files, the values are always read as strings, but you can convert them to different data types as needed. For example, you can process each row with a function that converts keys to the correct data types:

```
>>> def process_row(row):
...     row["salary"] = float(row["salary"])
...     return row
...
>>> with file_path.open(mode="r", encoding="utf-8", newline="") as file:
...     reader = csv.DictReader(file)
...     for row in reader:
...         print(process_row(row))
...
{'name': 'Lee', 'department': 'Operations', 'salary': 75000.0}
{'name': 'Jane', 'department': 'Engineering', 'salary': 85000.0}
{'name': 'Diego', 'department': 'Sales', 'salary': 80000.0}
```

The `process_row()` function takes a row dictionary read from the CSV file and returns a new dictionary with the "salary" key converted to a floating-point number.

You can write CSV files with headers using the `csv.DictWriter` class, which writes dictionaries with shared keys to rows in a CSV file.

The following list of dictionaries represents a small database of people and their ages:

```
>>> people = [
...     {"name": "Veronica", "age": 29},
...     {"name": "Audrey", "age": 32},
...     {"name": "Sam", "age": 24},
... ]
```

To store the data in the `people` list to a CSV file, open a new file called `people.csv` in write mode and create a new `csv.DictWriter` object from the `file` object:

```
>>> file_path = Path.home() / "people.csv"
>>> file = file_path.open(mode="w", encoding="utf-8", newline="")
>>> writer = csv.DictWriter(file, fieldnames=["name", "age"])
```

When you instantiate a new `DictWriter` object, the first parameter is the `file` object for writing the CSV data. The `fieldnames` parameter, which is required, is a list of strings of the field names.

Note

In the example above, the list literal `["name", "age"]` is passed to the `fieldnames` parameter.

You can also set `fieldnames` to `people[0].keys()` since `people[0]` is a dictionary whose keys are the field names. This is useful when the field names are unknown or when there are so many fields that a list literal is impractical.

Just like `csv.writer` objects, `DictWriter` objects have `.writerow()` and `.writerows()` methods for writing rows of data to the file. `DictWriter` objects also have a third method called `.writeheader()` that writes the header row to the CSV file:

```
>>> writer.writeheader()
10
```

`.writeheader()` returns the number of characters written to the file, which is 10 in this case. Writing the header row is optional but recommended because it helps define what the data contained in the CSV file represents. It also makes it easy to read the rows from the CSV file as dictionaries using the `DictReader` class.

With the header written, you can write the data in the `people` list to the CSV file using `.writerows()`:

```
>>> writer.writerows(people)
>>> file.close()
```

You now have a file in your home directory called `people.csv` containing the following data:

```
name,age
Veronica,29
Audrey,32
Sam,24
```

CSV files are a flexible and convenient way of storing data. They're frequently used in business worldwide, and knowing how to work with them is a valuable skill!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that writes the following list of lists to a file in your home directory called `numbers.csv`:

```
numbers = [
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15],
]
```

2. Write a program that reads the numbers in the `numbers.csv` file from exercise 1 into a list of lists of integers called `numbers`. Print the list of lists. Your output should look like this:

```
[[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]]
```

3. Write a program that writes the following list of dictionaries to a file in your home directory called `favorite_colors.csv`:

```
favorite_colors = [
    {"name": "Joe", "favorite_color": "blue"},
    {"name": "Anne", "favorite_color": "green"},
    {"name": "Bailey", "favorite_color": "red"},
]
```

The output CSV file should have the following format:

```
name,favorite color
Joe,blue
Anne,green
Bailey,red
```

4. Write a program that reads the data from the `favorite_colors.csv` file from exercise 3 into a list of dictionaries called `favorite_colors`. Print the list of dictionaries. The output should look something like this:

```
[{"name": "Joe", "favorite_color": "blue"},
 {"name": "Anne", "favorite_color": "green"},
 {"name": "Bailey", "favorite_color": "red"}]
```

12.7 Challenge: Create a High Scores List

In the chapter 12 `practice_files` folder, there is a CSV file called `scores.csv` containing data about game players and their scores. The first few lines of the file look like this:

```
name, score
LLCoolDave, 23
LLCoolDave, 27
red, 12
LLCoolDave, 26
tom123, 26
```

Write a program that reads the data from this CSV file and creates a new file called `high_scores.csv` in which each row contains the player's name and their highest score.

The output CSV file should look like this:

```
name, high_score
LLCoolDave, 27
red, 12
tom123, 26
O_O, 22
Misha46, 25
Empiro, 23
MaxxT, 25
L33tH4x, 42
johnsmith, 30
```

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

12.8 Summary and Additional Resources

In this chapter, you learned about the file system and file paths and how to work with them using the Python standard library’s `pathlib` module. You saw how to create new `Path` objects, how to access path components, and how to create, move, and delete files and folders.

You also learned how to read and write plain text files using the `Path.open()` method and built-in `open()` function as well as how to work with comma-separated value, or CSV, files using the Python standard library’s `csv` module.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-files

Additional Resources

To get even more practice working with files, check out these resources:

- “[Reading and Writing Files in Python \(Guide\)](#)”
- “[Working With Files in Python](#)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 13

Installing Packages With pip

Up to this point, you've been working within the bounds of the **Python standard library**. In the remaining half of this course, you'll work with various **packages** that aren't included with Python by default.

Many programming languages offer a **package manager** that automates the process of installing, upgrading, and removing third-party packages. Python is no exception.

The de facto package manager for Python is called `pip`. Historically, `pip` had to be downloaded and installed separately from Python. As of Python 3.4, it's now included with most distributions of the language.

In this chapter, you'll learn:

- How to install and manage third-party packages with `pip`
- What the benefits and risks of third-party packages are

Let's go!

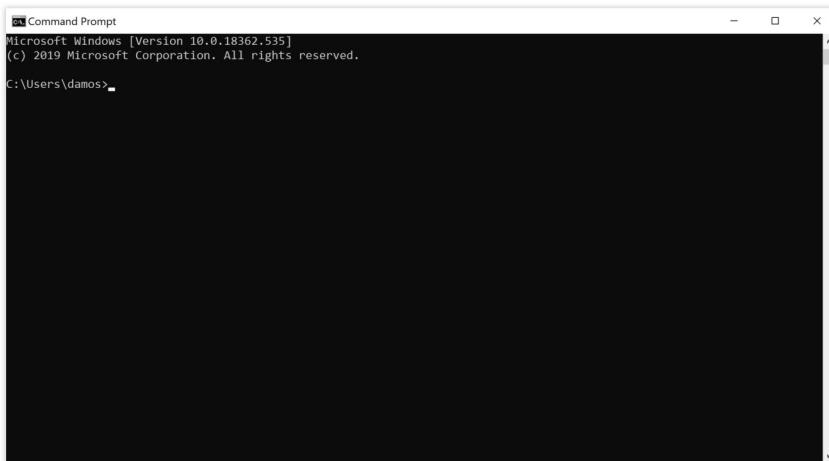
13.1 Installing Third-Party Packages With pip

Python’s package manager, `pip`, is used to install and manage third-party packages. It’s a separate program from Python, although it’s likely that `pip` was installed on your computer whenever you downloaded and installed Python.

`pip` is a **command-line tool**. That means you must run it from a command line or terminal program. How you open a terminal program depends on your operating system.

Windows

Press the Windows key, then type `cmd` and press `Enter` to open the Command Prompt application. This opens a window that looks like this:



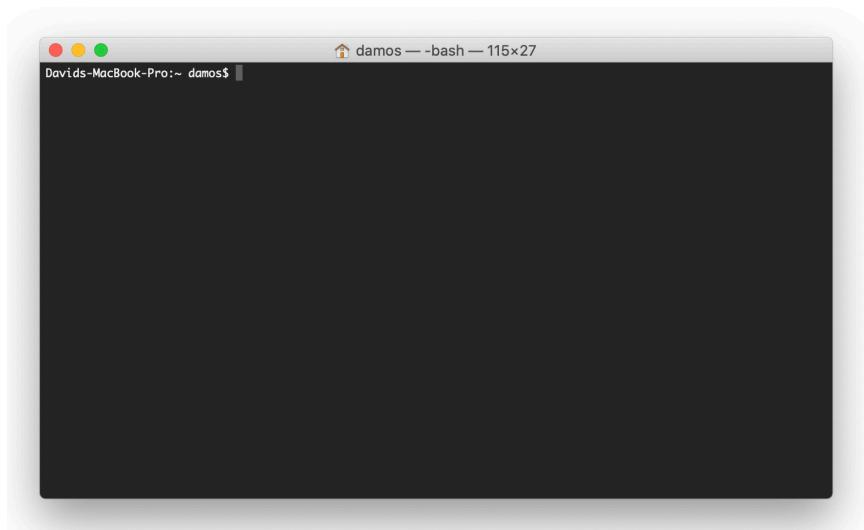
Alternatively, you may use the PowerShell application by pressing the Windows key, typing `powershell`, and pressing `Enter`. The PowerShell window looks like this:

13.1. Installing Third-Party Packages With pip



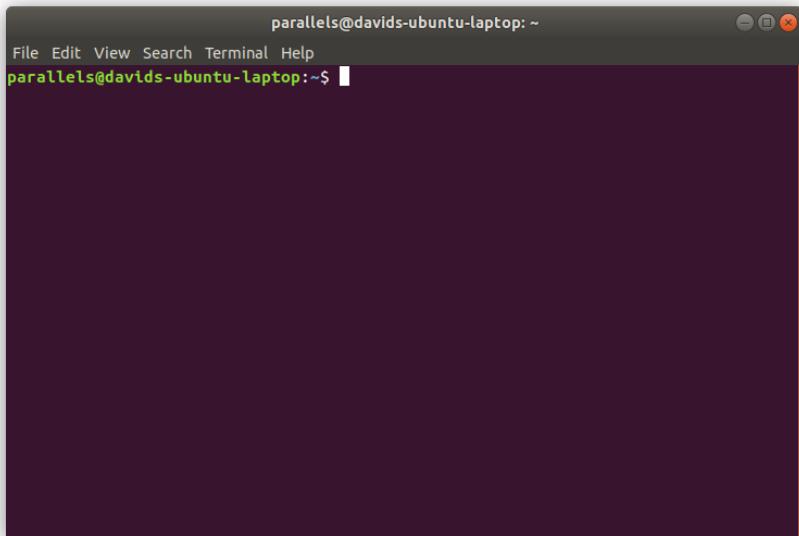
macOS

Press **Cmd** + **Spacebar** to open the Spotlight search window. Type **terminal** and press **Enter** to open the Terminal app. The window that opens look like this:



Ubuntu Linux

Click the `Show Applications` button at the bottom of your toolbar and search for `terminal`. Then click the Terminal application icon to open the terminal. The window that opens looks something like this:



Verify Your pip Installation

Now that you have a terminal open, let's verify that `pip` is installed on your computer. How you do this depends on your operating system.

On Windows, type the following command to check for `pip`:

```
$ python -m pip --version
```

For macOS and Linux, the following command can be used to verify that `pip` is installed:

```
$ python3 -m pip --version
```

If `pip` is installed, then you should see something like the following output displayed in your terminal:

```
pip 20.2.3 from c:\users\David\appdata\local\programs\python\  
python\lib\site-packages\pip (python 3.9)
```

This output indicates that version 20.2.3 of `pip` is currently installed and is linked to the Python 3.9 installation.

Important

If you don't see any output when verifying `pip`, or if you get an error, try running `python3.9 -m pip --version`. You might need to replace all future `python3` commands with `python3.9`.

Upgrading pip to the Latest Version

Before we go any further, let's make sure that you have the latest version of `pip` installed. To upgrade `pip`, type the following into your terminal and press `Enter`:

```
$ python3 -m pip install --upgrade pip
```

If a newer version of `pip` is available, then it will be downloaded and installed. Otherwise, you'll see a message indicating that the latest version is already installed. This message usually says something like `Requirement already satisfied.`

Important

In this section, we'll show all commands starting with `python3 -m pip`. This is important for macOS and Linux users, since the command `python -m pip` will possibly install packages for the wrong version of Python.

If you're a Windows user, however, you'll need to use `python -m pip`, since `python3 -m pip` will not work on your computer.

Now that you have `pip` upgraded to the latest version, let's see what you can do with it!

List All Installed Packages

You can use `pip` to list the packages you have installed. Let's take a peek at what's currently available. Type the following into your terminal:

```
$ python3 -m pip list
```

If you haven't installed any packages, which should be the case if you started this course with a fresh Python 3.9 installation, then you should see something like the following:

Package	Version
-----	-----
pip	19.3.1
setuptools	41.2.0

As you can see, there isn't much here. You see `pip` itself listed because `pip` is a package. You may also see `setuptools`. This is a package used by `pip` to setup and install other packages.

When you install a package with `pip`, it will show up in this list. You can always use `pip list` to see which packages, and which version of each package, you currently have installed.

Install a Package

Let's install your first Python package! For this exercise, you'll install the `requests` package, which is one of the most popular Python packages ever created.

In your terminal, type the following:

```
$ python3 -m pip install requests
```

While `pip` is installing the `requests` package, you'll see a bunch of output:

```
Collecting requests
  Downloading https://.../requests-2.22.0-py2.py3-none-any.whl (57kB)
    ..... | 61kB 2.0MB/s
Collecting urllib3!=1.25.0,!=1.25.1,<1.26,>=1.21.1
  Downloading https://...urllib3-1.25.7-py2.py3-none-any.whl (125kB)
    ..... | 133kB 3.3MB/s
Collecting certifi>=2017.4.17
  Downloading https://...certifi-2019.11.28.py3-none-any.whl (156kB)
    ..... | 163kB ...
Collecting chardet<3.1.0,>=3.0.2
  Downloading https://...chardet-3.0.4-py2.py3-none-any.whl (133kB)
    ..... | 143kB 6.8MB/s
Collecting idna<2.9,>=2.5
  Downloading https://...idna-2.8-py2.py3-none-any.whl (58kB)
    ..... | 61kB 3.8MB/s
Installing collected packages: urllib3, certifi, chardet, idna,
  requests
Successfully installed certifi-2019.11.28 chardet-3.0.4 idna-2.8
  requests-2.22.0 urllib3-1.25.7
```

Note

The formatting of the above output has been altered so that it fits nicely on the page. The output that you see on your computer may look different.

Notice that `pip` first tells you that it is “Collecting requests.” You’ll see the URL that `pip` is installing the package from, as well as a progress bar indicating the progress of the download.

After that, you’ll see that `pip` installs four more packages: `chardet`, `certifi`, `idna`, and `urllib3`. These packages are **dependencies** of `requests`. That means that `requests` requires these packages to be installed for it to work properly.

Once `pip` is done installing `requests` and its dependencies, run `pip list` in your terminal again. You should now see the following list:

```
$ python3 -m pip list
Package      Version
-----
certifi      2019.11.28
chardet      3.0.4
idna         2.8
pip          19.3.1
requests     2.22.0
setuptools   41.2.0
urllib3      1.25.7
```

You can see that version 2.22.0 of `requests` was installed, as well as the `chardet`, `certifi`, `idna`, and `urllib3` dependencies.

By default, `pip` installs the latest version of a package. You can control which version of a package gets installed with some optional version specifiers.

Install Specific Package Versions With Version Specifiers

There are several ways to control which version of a package gets installed. For example, you can:

1. Install the latest version greater than some version number
2. Install the latest version less than some version number
3. Install a specific version number

To install the latest version of `requests` whose version number is 2 or greater, you can execute the following:

```
$ python3 -m pip install requests>=2.0
```

Notice the `>=2.0` after the package name `requests`. This tells `pip` to install the latest version of `requests` that is greater than or equal to version 2.0.

The symbol `>=` is called a **version specifier** because it specifies which version of the package should be installed. There are several different

version specifiers that you can use. Here are the most widely used ones:

Version Specifier	Description
<code><=, >=</code>	Inclusive less than and greater than specifiers
<code><, ></code>	Exclusive less than and greater than specifiers
<code>==</code>	Exactly equal to specifier

Let's look at some examples.

To install the latest version that is less than or equal to some number, use the `<=` version specifier:

```
$ python3 -m pip install requests<=3.0
```

This will install the latest version of `requests` that is less than or equal to version 3.0.

The `<=` and `>=` version specifiers are **inclusive** because they include the version number that follows the specifier. **Exclusive** versions, `<` and `>`, exist as well.

For instance, the following command installs the latest version of `requests` that is strictly less than version 3.0:

```
$ python3 -m pip install requests<3.0
```

You can combine version specifiers to ensure `pip` installs the latest version within a specified version range. For example, the following command installs the latest version of `requests` in the 1.0 series:

```
$ python3 -m pip install requests>=1.0,<2.0
```

You would use something like the above command if your project was only compatible with the 1.0 series of the package and you want to make sure you install the latest updates to that series.

Finally, you can **pin** dependencies to a specific version with the `--version` specifier:

```
$ python3 -m pip install requests==2.22.0
```

This command installs exactly version 2.22.0 of the `requests` package.

Show Package Details

Now that you've installed the `requests` package, you can use `pip` to view some details about the package:

```
$ python3 -m pip show requests
Name: requests
Version: 2.22.0
Summary: Python HTTP for Humans.
Home-page: https://requests.readthedocs.io
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: c:\users\David\...\python\python\lib\site-packages
Requires: chardet, idna, certifi, urllib3
Required-by:
```

The `python3 -m pip show` command displays information about an installed package, including the author's name and email and a home page you can navigate to in your Internet browser to learn more about what the package does.

The `requests` package is used for making HTTP requests from a Python program. It's extremely useful in a variety of domains and is a dependency of a large number of other Python packages.

Uninstall a Package

If you can install a package with `pip`, then it only makes sense that you can also uninstall a package. Let's uninstall the `requests` package now.

To uninstall requests, type the following into your terminal:

```
$ python3 -m pip uninstall requests
```

Important

If you already have projects that use requests or one of its dependencies, then you may not want to run the commands in the remainder of this section.

You will immediately see the following prompt:

```
Uninstalling requests-2.22.0:  
Would remove:  
  c:\users\damos\...\requests-2.22.0.dist-info\*  
  c:\users\damos\...\requests\*  
Proceed (y/n)?
```

Before pip actually removes anything from your computer, it asks for your permission first. How considerate!

Type y and press **Enter** to continue. You should then see the following message confirming that requests was removed:

```
Successfully uninstalled requests-2.22.0
```

Take a look at your package list again:

```
$ python3 -m pip list  
Package      Version  
-----  
certifi      2018.4.16  
chardet      3.0.4  
idna         2.7  
pip          10.0.1  
setuptools   39.0.1  
urllib3      1.23
```

Notice that `pip` uninstalled `requests`, but it didn't remove any of its dependencies! This behavior is a feature, not a bug.

Imagine that you've installed several packages into your environment with `pip`, some of which share dependencies. If `pip` uninstalled a package *and* its dependencies, then it would render any other package requiring those dependencies unusable!

For now, though, go ahead and remove the remaining packages by running `pip uninstall`. You can uninstall all four in a single command:

```
$ python3 -m pip uninstall certifi chardet idna urllib3
```

When you're done, verify that everything has been removed by running `pip list` again. You should see the same list of packages you saw when you first started:

Package	Version
-----	-----
pip	10.0.1
setuptools	39.0.1

Python's ecosystem of third-party packages is one of its greatest strengths. These packages allow Python programmers to be highly productive and create full-featured software much more quickly than can be done in, say, a language like C++.

That said, using third-party packages in your code introduces several concerns that must be addressed with care. You'll learn about some of the pitfalls associated with third-party packages in the next section.

13.2 The Pitfalls of Third-Party Packages

The beauty of third-party packages is that they give you the ability to add functionality to your project without having to implement everything from scratch. This offers massive boosts in productivity.

But with great power comes great responsibility. As soon as you include someone else's package in your project, you are placing an enormous amount of trust in those responsible for developing and maintaining the package.

By using a package you did not develop, you lose control over certain aspects of your project. In particular, the maintainers of a package may release a new version that introduces changes that are incompatible with the version you use in your project.

By default, `pip` installs the latest release of a package, so if you distribute your code to someone else and they install a newer version of a package required by your project, then they may not be able to run your code.

This presents a significant challenge for both you and the end user. Fortunately, Python comes with a fix for this all-too-common problem: virtual environments.

A virtual environment creates an isolated and, most importantly, reproducible environment that you can use to develop a project. The environment can contain a specific version of Python as well as specific versions of your project's dependencies.

When you distribute your code to someone else, they can reproduce this environment and be confident that they can run your code without error.

Virtual environments are a more advanced topic that is outside the scope of this book. To learn more about virtual environments and how to use them, check out [Real Python's Managing Python Dependencies With Pip and Virtual Environments course](#). In it you'll learn how to:

- Install, use, and manage third-party Python packages with the `pip` package manager on Windows, macOS, and Linux in more detail than presented here
- Isolate project dependencies with virtual environments to avoid version conflicts in your Python projects

- Apply a complete seven-step workflow for finding and identifying quality third-party packages to use in your own Python projects (and for justifying your decisions to your team or manager)
- Set up repeatable development environments and application deployments using the `pip` package manager and requirements files

The [Managing Python Dependencies With Pip and Virtual Environments](#) course is a great next step for when you complete this book.

13.3 Summary and Additional Resources

In this chapter, you learned how to install third-party packages using Python’s package manager, `pip`. You saw several useful `pip` commands, including `pip install`, `pip list`, `pip show` and `pip uninstall`.

You also learned about some of the pitfalls associated with third-party packages. Not every package that is downloadable with `pip` is a good choice for your project. Since you don’t have control over the code in the package that you install, you must trust that the package is safe and will work well for the users of your program.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-installing-packages

Additional Resources

To learn more about managing third-party packages, you can check out these resources:

- [Managing Python Dependencies \(Course\)](#)

- “Python Virtual Environments: A Primer”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 14

Creating and Modifying PDF Files

The **PDF**, or Portable Document Format, is one of the most common formats for sharing documents over the Internet. PDFs can contain text, images, tables, forms, and rich media like videos and animations, all in a single file.

This abundance of content types can make working with PDFs difficult. There are a lot of different kinds of data to decode when opening a PDF file! Fortunately, the Python ecosystem has some great packages for reading, manipulating, and creating PDF files.

In this chapter, you'll learn how to:

- Read text from a PDF
- Split a PDF into multiple files
- Concatenate and merge PDF files
- Rotate and crop pages in a PDF file
- Encrypt and decrypt PDF files with passwords
- Create a PDF file from scratch

Let's get started!

14.1 Extracting Text From a PDF

In this section, you'll learn how to read a PDF file and extract the text using the [PyPDF2](#) package. Before you can do that, though, you need to install it with pip:

```
$ python3 -m pip install PyPDF2
```

Verify the installation by running the following command in your terminal:

```
$ python3 -m pip show PyPDF2
Name: PyPDF2
Version: 1.26.0
Summary: PDF toolkit
Home-page: http://mstamy2.github.com/PyPDF2
Author: Mathieu Fenniak
Author-email: biziqe@mathieu.fenniak.net
License: UNKNOWN
Location: c:\users\david\python\lib\site-packages
Requires:
Required-by:
```

Pay particular attention to the version information. At the time of writing, the latest version of PyPDF2 was 1.26.0. If you have IDLE open, then you'll need to restart it before you can use the PyPDF2 package.

Opening a PDF File

Let's get started by opening a PDF and reading some information about it. You'll use the `Pride_and_Prejudice.pdf` file located in the chapter 14 `practice_files` folder.

Note

If you haven't already, you can download the exercise solutions and practice files at the following URL:

<https://github.com/realpython/python-basics-exercises>

Open IDLE's interactive window and import the `PdfFileReader` class from the `PyPDF2` package:

```
>>> from PyPDF2 import PdfFileReader
```

To create a new instance of the `PdfFileReader` class, you'll need the path to the PDF file that you want to open. Let's get that now using the `pathlib` module:

```
>>> from pathlib import Path
>>> pdf_path = (
...     Path.home() /
...     "python-basics-exercises" /
...     "ch14-interact-with-pdf-files" /
...     "practice_files" /
...     "Pride_and_Prejudice.pdf"
... )
```

The `pdf_path` variable now contains the path to a PDF version of Jane Austen's *Pride and Prejudice*.

Note

You may need to change `pdf_path` so that it corresponds to the location of the `python-basics-exercises/` folder on your computer.

Now create the `PdfFileReader` instance:

```
>>> pdf = PdfFileReader(str(pdf_path))
```

You convert `pdf_path` to a string because `PdfFileReader` doesn't know how to read from a `pathlib.Path` object.

Recall from chapter 12, “File Input and Output,” that all open files should be closed before a program terminates. The `PdfFileReader` object does all of this for you, so you don’t need to worry about opening or closing the PDF file!

Now that you’ve created a `PdfFileReader` instance, you can use it to gather information about the PDF. For example, `.getNumPages()` returns the number of pages contained in the PDF file:

```
>>> pdf.getNumPages()  
234
```

You may have noticed that `.getNumPages()` is written in `mixedCase` and not in `lower_case_with_underscores` as recommended in [PEP 8](#). Remember, PEP 8 is a set of guidelines, not rules. As far as Python is concerned, `mixedCase` is perfectly acceptable.

Note

PyPDF2 was adapted from the `pyPdf` package. `pyPdf` was written in 2005, only four years after PEP 8 was published.

At that time, many Python programmers were migrating from languages in which `mixedCase` was more common.

You can also access some document information using the `.documentInfo` attribute:

```
>>> pdf.documentInfo  
{'/Title': 'Pride and Prejudice, by Jane Austen', '/Author': 'Chuck',  
'/Creator': 'Microsoft® Office Word 2007',  
'/CreationDate': 'D:20110812174208', '/ModDate': 'D:20110812174208',  
'/Producer': 'Microsoft® Office Word 2007'}
```

The object returned by `.documentInfo` looks like a dictionary, but it’s not really the same thing. You can access each item in `.documentInfo` as an attribute.

For example, to get the title, use the `.title` attribute:

```
>>> pdf.documentInfo.title  
'Pride and Prejudice, by Jane Austen'
```

The `.documentInfo` object contains the PDF **metadata**, which is set when a PDF is created.

The `PdfFileReader` class provides all the necessary methods and attributes that you need to access data in a PDF file. Let's explore what you can do with a PDF file and how you can do it!

Extracting Text From a Page

PDF pages are represented in `PyPDF2` with the `PageObject` class. You use `PageObject` instances to interact with pages in a PDF file. You don't need to create your own `PageObject` instances directly. Instead, you can access them through the `PdfFileReader` object's `.getPage()` method.

There are two steps to extracting text from a single PDF page:

1. Get a `PageObject` with `PdfFileReader.getPage()`.
2. Extract the text as a string with the `PageObject` instance's `.extractText()` method.

`Pride_and_Prejudice.pdf` has 234 pages. Each page has an index between 0 and 233. You can get the `PageObject` representing a specific page by passing the page's index to `PdfFileReader.getPage()`:

```
>>> first_page = pdf.getPage(0)
```

`.getPage()` returns a `PageObject`:

```
>>> type(first_page)  
<class 'PyPDF2.pdf.PageObject'>
```

You can extract the page's text with `PageObject.extractText()`:

```
>>> first_page.extractText()
'\n \nThe Project Gutenberg EBook of Pride and Prejudice, by Jane
Austen\n \n\nThis eBook is for the use of anyone anywhere at no cost
and with\n \nalmost no restrictions whatsoever. You may copy it,
give it away or\n \nre\n-\nuse it under the terms of the Project
Gutenberg License included\n \nwith this eBook or online at
www.gutenberg.org\n \n \n\nTitle: Pride and Prejudice\n \n
\nAuthor: Jane Austen\n \n \nRelease Date: August 26, 2008
[EBook #1342]\n\n[Last updated: August 11, 2011]\n \n \nLanguage:
Eng\nlish\n \n \nCharacter set encoding: ASCII\n \n \n***\n
START OF THIS PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***\n \n
\n \n \n \n\nProduced by Anonymous Volunteers, and David Widger\n
\n \n \n \n \n \n \nPRIDE AND PREJUDICE \n \n \n\nBy Jane
Austen \n \n\n \n \nContents\n \n '
```

Note that this output has been formatted to fit better on the page. The output you see on your computer may be formatted differently.

Note

Depending on how a PDF is encoded, text read from the PDF might include unexpected characters or may not include line breaks. This is one of the downsides to reading text from a PDF.

In the real world, you may find yourself doing some manual cleanup when reading text from a PDF.

Every `PdfFileReader` object has a `.pages` attribute that you can use to iterate over all of the pages in the PDF in order.

For example, the following `for` loop prints the text from every page in the *Pride and Prejudice* PDF:

```
>>> for page in pdf.pages:
...     print(page.extractText())
...
```

Let's combine everything you've learned and write a program that extracts all of the text from the `Pride_and_Prejudice.pdf` file and saves it to a `.txt` file.

Putting It All Together

Open a new editor window in IDLE and type in the following code:

```
from pathlib import Path
from PyPDF2 import PdfFileReader

# Change the path below to the correct path for your computer.
pdf_path = (
    Path.home() /
    "python-basics-exercises" /
    "ch14-interact-with-pdf-files" /
    "practice-files" /
    "Pride_and_Prejudice.pdf"
)

# 1
pdf_reader = PdfFileReader(str(pdf_path))
output_file_path = Path.home() / "Pride_and_Prejudice.txt"

# 2
with output_file_path.open(mode="w") as output_file:
    # 3
    title = pdf_reader.getDocumentInfo.title
    num_pages = pdf_reader.getNumPages()
    output_file.write(f"{title}\nNumber of pages: {num_pages}\n\n")

# 4
for page in pdf_reader.pages:
    text = page.extractText()
    output_file.write(text)
```

Let's break that down:

1. First, you assign a new `PdfFileReader` instance to the `pdf_reader` variable. You also create a new `Path` object that points to the file `Pride_and_Prejudice.txt` in your home directory and assign it to the `output_file_path` variable.
2. Next, you open `output_file_path` in write mode and assign the file object returned by `.open()` to the variable `output_file`. The `with` statement, which you learned about in chapter 12, “File Input and Output,” ensures that the file is closed when the `with` block exits.
3. Then, inside the `with` block, you write the PDF title and number of pages to the text file using `output_file.write()`.
4. Finally, you use a `for` loop to iterate over all the pages in the PDF. At each step in the loop, the next `PageObject` is assigned to the `page` variable. The text from each page is extracted with `page.extractText()` and is written to the `output_file`.

When you save and run the program, it will create a new file in your home directory called `Pride_and_Prejudice.txt` containing the full text of the `Pride_and_Prejudice.pdf` document. Open it up and check it out!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. The chapter 14 `practice_files` folder contains a PDF file called `zen.pdf`. Create a `PdfFileReader` instance from this PDF.
2. Using the `PdfFileReader` instance from exercise 1, print the total number of pages in the PDF.
3. Print the text from the first page of the PDF in exercise 1.

14.2 Extracting Pages From a PDF

In the previous section, you learned how to extract all of the text from a PDF file and save it to a .txt file. Now you'll learn how to extract a page or range of pages from an existing PDF and save them to a new PDF.

You can use the `PdfFileWriter` to create a new PDF file. Let's explore this class and learn the steps needed to create a PDF using `PyPDF2`.

Using the `PdfFileWriter` Class

The `PdfFileWriter` class creates new PDF files. In IDLE's interactive window, import the `PdfFileWriter` class and create a new instance called `pdf_writer`:

```
>>> from PyPDF2 import PdfFileWriter  
>>> pdf_writer = PdfFileWriter()
```

`PdfFileWriter` objects are like blank PDF files. You need to add some pages to them before you can save them to a file. Go ahead and add a blank page to `pdf_writer`:

```
>>> page = pdf_writer.addBlankPage(width=72, height=72)
```

The `width` and `height` parameters are required and determine the dimensions of the page in units called **points**. One point equals $1/72$ of an inch, so the above code adds a one-inch-square blank page to `pdf_writer`.

`.addBlankPage()` returns a new `PageObject` instance representing the page that you added to the `PdfFileWriter`:

```
>>> type(page)  
<class 'PyPDF2.pdf.PageObject'>
```

In this example, you've assigned the `PageObject` instance returned by `.addBlankPage()` to the `page` variable, but in practice you don't usually

need to do this. That is, you usually call `.addBlankPage()` without assigning the return value to anything:

```
>>> pdf_writer.addBlankPage(width=72, height=72)
```

To write the contents of `pdf_writer` to a PDF file, pass a file object in binary write mode to `pdf_writer.write()`:

```
>>> from pathlib import Path
>>> with Path("blank.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
>>>
```

This creates a new file in your current working directory called `blank.pdf`. If you open the file with a PDF reader, such as Adobe Acrobat, then you'll see a document with a single blank one-inch-square page.

Important

Notice that you save the PDF file by passing the file object to the `PdfFileWriter` object's `.write()` method, *not* to the file object's `.write()` method.

In particular, the following code will not work:

```
>>> with Path("blank.pdf").open(mode="wb") as output_file:
...     output_file.write(pdf_writer)
```

This approach seems backward to many new programmers, so make sure you avoid this mistake!

`PdfFileWriter` objects can write to new PDF files, but they can't create new content from scratch other than blank pages.

This might seem like a big problem, but in many situations, you don't need to create new content. Often, you'll work with pages extracted from PDF files that you've opened with a `PdfFileReader` instance.

Note

You'll learn how to create PDF files from scratch in section 14.8, "Creating a PDF File From Scratch."

In the example you saw above, there were three steps to create a new PDF file using `PyPDF2`:

1. Create a `PdfFileWriter` instance.
2. Add one or more pages to the `PdfFileWriter` instance.
3. Write to a file using `PdfFileWriter.write()`.

You'll see this pattern over and over as you learn various ways to add pages to a `PdfFileWriter` instance.

Extracting a Single Page From a PDF

Let's revisit the *Pride and Prejudice* PDF that you worked with in the previous section. You'll open the PDF, extract the first page, and create a new PDF file containing just the single extracted page.

Open IDLE's interactive window and import `PdfFileReader` and `PdfFileWriter` from `PyPDF2` as well as the `Path` class from the `pathlib` module:

```
>>> from pathlib import Path  
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now open the `Pride_and_Prejudice.pdf` file with a `PdfFileReader` instance:

```
>>> # Change the path to work on your computer if necessary  
>>> pdf_path = (  
...     Path.home() /  
...     "python-basics-exercises" /  
...     "ch14-interact-with-pdf-files" /  
...     "practice_files" /  
...     "Pride_and_Prejudice.pdf"  
... )  
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Pass the index 0 to `.getPage()` to get a `PageObject` representing the first page of the PDF:

```
>>> first_page = input_pdf.getPage(0)
```

Now create a new `PdfFileWriter` instance and add `first_page` to it with `.addPage()`:

```
>>> pdf_writer = PdfFileWriter()  
>>> pdf_writer.addPage(first_page)
```

The `.addPage()` method adds a page to the set of pages in the `pdf_writer` object, just like `.addBlankPage()`. The difference is that it requires an existing `PageObject`.

Now write the contents of `pdf_writer` to a new file:

```
>>> with Path("first_page.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...  
>>>
```

You now have a new PDF file called `first_page.pdf` saved in your current working directory that contains the cover page of the *Pride and Prejudice* PDF file. Pretty neat!

Extracting Multiple Pages From a PDF

Let's extract the first chapter from `Pride_and_Prejudice.pdf` and save it to a new PDF.

If you open `Pride_and_Prejudice.pdf` with a PDF viewer, then you can see that the first chapter is on the second, third, and fourth pages of the PDF. Since pages are indexed starting with 0, you'll need to extract the pages at the indices 1, 2, and 3.

You can set everything up by importing the classes you need and opening the PDF file:

```
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> from pathlib import Path
>>> pdf_path = (
...     Path.home() /
...     "python-basics-exercises" /
...     "ch14-interact-with-pdf-files" /
...     "practice_files" /
...     "Pride_and_Prejudice.pdf"
... )
>>> input_pdf = PdfFileReader(str(pdf_path))
```

Your goal is to extract the pages at indices 1, 2, and 3, add these to a new `PdfFileWriter` instance, and then write them to a new PDF file.

One way to do this is to loop over the range of numbers starting at 1 and ending at 3, extracting the page at each step of the loop and adding it to the `PdfFileWriter` instance:

```
>>> pdf_writer = PdfFileWriter()
>>> for n in range(1, 4):
...     page = input_pdf.getPage(n)
...     pdf_writer.addPage(page)
...
>>>
```

The loop iterates over the numbers 1, 2, and 3 since `range(1, 4)` doesn't include the right-hand endpoint. At each step in the loop, the page at the current index is extracted with `.getPage()` and then added to the `pdf_writer` using `.addPage()`.

Now `pdf_writer` has three pages, which you can check with `.getNumPages()`:

```
>>> pdf_writer.getNumPages()  
3
```

Finally, you can write the extracted pages to a new PDF file:

```
>>> with Path("chapter1.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...  
>>>
```

Now you can open the `chapter1.pdf` file in your current working directory to read just the first chapter of *Pride and Prejudice*.

Another way to extract multiple pages from a PDF is to take advantage of the fact that `PdfFileReader.pages` supports slice notation. Let's redo the previous example using `.pages` instead of looping over a `range` object.

Start by initializing a new `PdfFileWriter` object:

```
>>> pdf_writer = PdfFileWriter()
```

Now loop over a slice of `.pages` from indices starting at 1 and ending at 4:

```
>>> for page in input_pdf.pages[1:4]:  
...     pdf_writer.addPage(page)  
...  
>>>
```

Remember that the values in a slice range from the item at the first index in the slice up to but not including the item at the second index in the slice. So `.pages[1:4]` returns an iterable containing the pages with indices 1, 2, and 3.

Finally, write the contents of `pdf_writer` to the output file:

```
>>> with Path("chapter1_slice.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...  
>>>
```

Now open the `chapter1_slice.pdf` file in your current working directory and compare it to the `chapter1.pdf` file you made by looping over the `range` object. They contain the same pages!

Sometimes you need to extract every page from a PDF. You can use the methods illustrated above to do this, but `PyPDF2` provides a shortcut. `PdfFileWriter` instances have an `.appendPagesFromReader()` method that you can use to append pages from a `PdfFileReader` instance.

To use `.appendPagesFromReader()`, pass a `PdfFileReader` instance to the method's `reader` parameter. For example, the following code copies every page from the *Pride and Prejudice* PDF to a `PdfFileWriter` instance:

```
>>> pdf_writer = PdfFileWriter()  
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

`pdf_writer` now contains every page in `pdf_reader`!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Extract the last page from the `Pride_and_Prejudice.pdf` file and save it to a new file in your home directory called `last_page.pdf`.

2. Extract all pages with even-numbered indices (not page numbers) from the `Pride_and_Prejudice.pdf` and save them to a new file in your home directory called `every_other_page.pdf`.
3. Split the `Pride_and_Prejudice.pdf` file into two new PDF files. The first file should contain the first 150 pages, and the second file should contain the remaining pages. Save both files in your home directory as `part_1.pdf` and `part_2.pdf`.

14.3 Challenge: PdfFileSplitter Class

Create a class called `PdfFileSplitter` that reads a PDF from an existing `PdfFileReader` instance and splits the PDF into two new PDFs. The class should be instantiated with a path string.

For example, here's how you would create a `PdfFileSplitter` instance from a PDF in your current working directory called `mydoc.pdf`:

```
pdf_splitter = PdfFileSplitter("mydoc.pdf")
```

The `PdfFileSplitter` class should have two methods:

1. `.split()`, which has a single parameter, `breakpoint`, that expects an integer representing the page number at which to split the PDF.
2. `.write()`, which has a single parameter, `filename`, that expects a path string.

After you call `.split()`, the `PdfFileSplitter` class should have the attribute `.writer1` assigned to a `PdfFileWriter` instance containing all the pages in the original PDF up to *but not including* the breakpoint page. It should also have the attribute `.writer2` assigned to a `PdfFileWriter` instance containing the remaining pages in the original PDF.

When you call `.write()`, two PDFs should be written to the specified path, the first with the name `filename + "_1.pdf"` and the second with the name `filename + "_2.pdf"`.

For example, here's how you would split the `mydoc.pdf` at page four and write to two files called `mydoc_split_1.pdf` and `mydoc_split_2.pdf`:

```
pdf_splitter.split(breakpoint=4)  
pdf_splitter.write("mydoc_split")
```

Check that the splitter works by splitting the `Pride_and_Prejudice.pdf` file in the chapter 14 `practice_files` folder with the breakpoint at page 150.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

14.4 Concatenating and Merging PDFs

Two common tasks when working with PDF files are concatenating and merging several PDFs into a single file.

When you **concatenate** two or more PDFs, you join the files one after another into a single document. For example, a company may concatenate several daily reports into one monthly report at the end of a month.

Merging two PDFs also joins the PDFs into a single file. But instead of joining the second PDF to the end of the first, merging allows you to insert it after a specific page in the first PDF. Then it pushes all of the first PDF's pages after the insertion point to the end of the second PDF.

In this section, you'll learn how to concatenate and merge PDFs using the `PyPDF2` package's `PdfFileMerger`.

Using the `PdfFileMerger` Class

The `PdfFileMerger` class is a lot like the `PdfFileWriter` class that you learned about in the previous section. You can use both classes to write PDF files. In both cases, you add pages to instances of the class and then write them to a file.

The main difference between the two is that `PdfFileWriter` can only append, or concatenate, pages to the end of the list of pages already contained in the writer, whereas `PdfFileMerger` can insert, or merge, pages at any location.

Go ahead and create your first `PdfFileMerger` instance. In IDLE's interactive window, type the following code to import the `PdfFileMerger` class and create a new instance:

```
>>> from PyPDF2 import PdfFileMerger  
>>> pdf_merger = PdfFileMerger()
```

`PdfFileMerger` objects are empty when they're first instantiated. You'll need to add some pages to your object before you can do anything with it.

There are a couple of ways to add pages to the `pdf_merger` object, and which one you use depends on what you need to accomplish:

- `.append()` concatenates every page in an existing PDF document to the end of the pages currently in the `PdfFileMerger`.
- `.merge()` inserts all of the pages in an existing PDF document after a specific page in the `PdfFileMerger`.

You'll look at both methods in this section, starting with `.append()`.

Concatenating PDFs With `.append()`

The chapter `14 practice_files` folder has a subdirectory called `expense_reports` that contains three expense reports for an employee named Peter Python.

Peter needs to concatenate these three PDFs and submit them to his employer as a single PDF file so that he can get reimbursed for some work-related expenses.

You can start by using the `pathlib` module to get a list of `Path` objects for each of the three expense reports in the `expense_reports/` folder:

```
>>> from pathlib import Path  
>>> reports_dir = (  
...     Path.home() /  
...     "python-basics-exercises" /  
...     "ch14-interact-with-pdf-files" /  
...     "practice_files" /  
...     "expense_reports"  
... )
```

After you import the `Path` class, you need to build the path to the `expense_reports/` directory. Note that you may need to alter the code above to get the correct path on your computer.

Once you have the path to the `expense_reports/` directory assigned to the `reports_dir` variable, you can use `.glob()` to get an iterable of paths to PDF files in the directory.

Take a look at what's in the directory:

```
>>> for path in reports_dir.glob("*.pdf"):  
...     print(path.name)  
...  
Expense report 1.pdf  
Expense report 3.pdf  
Expense report 2.pdf
```

The names of the three files are listed, but they aren't in order. Furthermore, the order of the files you see in the output on your computer may not match the output shown here.

In general, the order of paths returned by `.glob()` is not guaranteed, so you'll need to order them yourself. You can do this by creating a list containing the three file paths and then calling `.sort()` on that list:

```
>>> expense_reports = list(reports_dir.glob("*.pdf"))  
>>> expense_reports.sort()
```

Remember that `.sort()` sorts a list in place, so you don't need to assign the return value to a variable. The `expense_reports` list will be sorted alphabetically by filename after `.list()` is called.

To confirm that the sorting worked, loop over `expense_reports` again and print out the filenames:

```
>>> for path in expense_reports:  
...     print(path.name)  
...  
Expense report 1.pdf  
Expense report 2.pdf  
Expense report 3.pdf
```

That looks good!

Now you can concatenate the three PDFs. To do that, you'll use `PdfFileMerger.append()`, which requires a single string argument representing the path to a PDF file. When you call `.append()`, all of the pages in the PDF file are appended to the set of pages in the `PdfFileMerger` object.

Let's see this in action. First, import the `PdfFileMerger` class and create a new instance:

```
>>> from PyPDF2 import PdfFileMerger  
>>> pdf_merger = PdfFileMerger()
```

Now loop over the paths in the sorted `expense_reports` list and append them to `pdf_merger`:

```
>>> for path in expense_reports:  
...     pdf_merger.append(str(path))  
...  
>>>
```

Notice that each `Path` object in `expense_reports` is converted to a string with `str()` before being passed to `pdf_merger.append()`.

With all the PDF files in the `expense_reports/` directory concatenated in the `pdf_merger` object, the last thing you need to do is write everything to an output PDF file. `PdfFileMerger` instances have a `.write()` method that works just like the `PdfFileWriter.write()`.

Open a new file in binary write mode, then pass the file object to the `pdf_merge.write()` method:

```
>>> with Path("expense_reports.pdf").open(mode="wb") as output_file:  
...     pdf_merger.write(output_file)  
...  
>>>
```

You now have a PDF file in your current working directory called `expense_reports.pdf`. Open it up with a PDF reader and you'll find all three expense reports together in the same PDF file.

Merging PDFs With `.merge()`

To merge two or more PDFs, use `PdfFileMerger.merge()`. This method is similar to `.append()`, except that you must specify where in the output PDF to insert all of the content from the PDF you are merging.

Take a look at an example. Goggle, Inc. prepared a quarterly report but forgot to include a table of contents. Peter Python noticed the mistake and quickly created a PDF with the missing table of contents. Now he needs to merge that PDF into the original report.

Both the report PDF and the table of contents PDF can be found in the `quarterly_report/` subfolder of the chapter `14 practice_files` folder. The report is in a file called `report.pdf`, and the table of contents is in a file called `toc.pdf`.

In IDLE's interactive window, import the `PdfFileMerger` class and create the `Path` objects for the `report.pdf` and `toc.pdf` files:

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileMerger
>>> report_dir = (
...     Path.home() /
...     "python-basics-exercises" /
...     "ch14-interact-with-pdf-files" /
...     "practice_files" /
...     "quarterly_report"
... )
>>> report_path = report_dir / "report.pdf"
>>> toc_path = report_dir / "toc.pdf"
```

The first thing you'll do is append the report PDF to a new `PdfFileMerger` instance using `.append()`:

```
>>> pdf_merger = PdfFileMerger()
>>> pdf_merger.append(str(report_path))
```

Now that `pdf_merger` has some pages in it, you can merge the table of contents PDF into it at the correct location. If you open the `report.pdf` file with a PDF reader, then you'll see that the first page of the report is a title page. The second is an introduction, and the remaining pages contain different report sections.

You want to insert the table of contents after the title page and just before the introduction section. Since PDF page indices start with 0 in `PyPDF2`, you need to insert the table of contents after the page at index 0 and before the page at index 1.

To do that, call `pdf_merger.merge()` with two arguments:

1. The integer 1, indicating the index of the page at which the table of contents should be inserted
2. A string containing the path of the PDF file for the table of contents

Here's what that looks like:

```
>>> pdf_merger.merge(1, str(toc_path))
```

Every page in the table of contents PDF is inserted *before* the page at index 1. Since the table of contents PDF is only one page, it gets inserted at index 1. The page currently at index 1 then gets shifted to index 2. The page currently at index 2 gets shifted to index 3, and so on.

Now write the merged PDF to an output file:

```
>>> with Path("full_report.pdf").open(mode="wb") as output_file:  
...     pdf_merger.write(output_file)  
...  
>>>
```

You now have a `full_report.pdf` file in your current working directory. Open it up with a PDF reader and check that the table of contents was inserted at the correct spot.

Concatenating and merging PDFs are common operations. While the examples in this section are admittedly somewhat contrived, you can imagine how useful a program would be for merging thousands of PDFs or for automating routine tasks that would otherwise take a human lots of time to complete.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. The chapter 14 `practice_files` folder contains three PDFs called `merge1.pdf`, `merge2.pdf`, and `merge3.pdf`. Using a `PdfFileMerger` instance, concatenate the two files `merge1.pdf` and `merge2.pdf` using `.append()`.

Save the concatenated PDFs to a file called `concatenated.pdf` in your home directory.

2. With a new `PdfFileMerger` instance, use `.merge()` to merge the file `merge3.pdf` between the two pages in the `concatenated.pdf` file you made in exercise 1. Save the new file to your home directory as `merged.pdf`.

The final result should be a PDF with three pages. The first page should have the number 1 on it, the second should have 2, and the third should have 3.

14.5 Rotating and Cropping PDF Pages

So far, you've learned how to extract text and pages from PDFs and how to and concatenate and merge two or more PDF files. These are all common operations with PDFs, but `PyPDF2` has many other useful features.

In this section, you'll learn how to rotate and crop pages in a PDF file.

Rotating Pages

You'll start by learning how to rotate pages. For this example, you'll use the `ugly.pdf` file in the chapter 14 `practice_files` folder.

The `ugly.pdf` file contains a lovely version of Hans Christian Andersen's *The Ugly Duckling*, except that every odd-numbered page is rotated counterclockwise by ninety degrees.

Let's fix that. In a new IDLE interactive window, start by importing the `PdfFileReader` and `PdfFileWriter` classes from `PyPDF2`, as well as the `Path` class from the `pathlib` module:

```
>>> from pathlib import Path  
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a `Path` object for the `ugly.pdf` file:

```
>>> pdf_path = (
...     Path.home() /
...     "python-basics-exercises" /
...     "ch14-interact-with-pdf-files" /
...     "practice_files" /
...     "ugly.pdf"
... )
```

Finally, create new `PdfFileReader` and `PdfFileWriter` instances:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Your goal is to use `pdf_writer` to create a new PDF file in which all of the pages have the correct orientation. The even-numbered pages in the PDF are already properly oriented, but the odd-numbered pages are rotated counterclockwise by ninety degrees.

To correct the problem, you'll use `PageObject.rotateClockwise()`. This method takes an integer argument, in degrees, and rotates a page clockwise by that many degrees. For example, `.rotateClockwise(90)` rotates a PDF page clockwise by ninety degrees.

Note

In addition to `.rotateClockwise()`, the `PageObject` class also has `.rotateCounterClockwise()` for rotating pages counterclockwise.

There are several ways you can go about rotating pages in the PDF. We'll discuss two different ways of doing it. Both of them rely on `.rotateClockwise()`, but they take different approaches to determine which pages get rotated.

The first technique is to loop over the indices of the pages in the PDF and check if each index corresponds to a page that needs to be rotated. If so, then you'll call `.rotateClockwise()` to rotate the page and then add the page to `pdf_writer`.

Here's what that looks like:

```
>>> for n in range(pdf_reader.getNumPages()):
...     page = pdf_reader.getPage(n)
...     if n % 2 == 0:
...         page.rotateClockwise(90)
...     pdf_writer.addPage(page)
...
>>>
```

Notice that the page gets rotated if the index is even. That might seem strange since the odd-numbered pages in the PDF are the ones that are rotated incorrectly. However, the page numbers in the PDF start with 1, whereas the page indices start with 0. That means odd-numbered PDF pages have even indices.

If that makes your head spin, don't worry! Even after years of dealing with stuff like this, professional programmers still get tripped up by these sorts of things!

Note

When you execute the `for` loop above, you'll see a bunch of output in IDLE's interactive window. That's because `.rotateClockwise()` returns a `PageObject` instance.

You can ignore this output for now. When you execute programs from IDLE's editor window, this output won't be visible.

Now that you've rotated all the pages in the PDF, you can write the content of `pdf_writer` to a new file and check that everything worked:

```
>>> with Path("ugly_rotated.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
>>>
```

You should now have a file in your current working directory called `ugly_rotated.pdf`, with the pages from the `ugly.pdf` file rotated correctly.

The problem with the approach you just used to rotate the pages in the `ugly.pdf` file is that it depends on knowing ahead of time which pages need to be rotated. In a real-world scenario, it isn't practical to go through an entire PDF taking note of which pages to rotate.

In fact, you can determine which pages need to be rotated without prior knowledge. Well, *sometimes* you can.

Let's see how, starting with a new `PdfFileReader` instance:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

You need to do this because you altered the pages in the old `PdfFileReader` instance by rotating them. So, by creating a new instance, you're starting fresh.

`PageObject` instances maintain a dictionary of values containing information about the page:

```
>>> pdf_reader.getPage(0)
{'/Contents': [IndirectObject(11, 0), IndirectObject(12, 0),
IndirectObject(13, 0), IndirectObject(14, 0), IndirectObject(15, 0),
IndirectObject(16, 0), IndirectObject(17, 0), IndirectObject(18, 0)],
'/Rotate': -90, '/Resources': {'/ColorSpace': {'/CS1':
IndirectObject(19, 0), '/CS0': IndirectObject(19, 0)}, '/XObject':
{'/Im0': IndirectObject(21, 0)}, '/Font': {'/TT1':
IndirectObject(23, 0), '/TTO': IndirectObject(25, 0)}, '/ExtGState':
{'/GS0': IndirectObject(27, 0)}}, '/CropBox': [0, 0, 612, 792],
'/Parent': IndirectObject(1, 0), '/MediaBox': [0, 0, 612, 792],
'/Type': '/Page', '/StructParents': 0}
```

Yikes! Mixed in with all that nonsensical-looking stuff is a key called `/Rotate`, which you can see on the fourth line of output above. The value of this key is `-90`.

You can access the `/Rotate` key on a `PageObject` using subscript notation, just like you can on a Python `dict` object:

```
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
```

If you look at the `/Rotate` key for the second page in `pdf_reader`, you'll see that it has a value of 0:

```
>>> page = pdf_reader.getPage(1)
>>> page["/Rotate"]
0
```

What all this means is that the page at index 0 has a rotation value of -90 degrees. In other words, it's been rotated counterclockwise by ninety degrees. The page at index 1 has a rotation value of 0, so it has not been rotated at all.

If you rotate the first page using `.rotateClockwise()`, then the value of `/Rotate` changes from -90 to 0:

```
>>> page = pdf_reader.getPage(0)
>>> page["/Rotate"]
-90
>>> page.rotateClockwise(90)
>>> page["/Rotate"]
0
```

Now that you know how to inspect the `/Rotate` key, you can use it to rotate the pages in the `ugly.pdf` file.

The first thing you need to do is to reinitialize both the `pdf_reader` and the `pdf_writer` objects so that you get a fresh start:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now write a loop that loops over the pages in the `pdf_reader.pages` iterable, checks the value of `/Rotate`, and rotates the page if that value is `-90`:

```
>>> for page in pdf_reader.pages:  
...     if page["/Rotate"] == -90:  
...         page.rotateClockwise(90)  
...     pdf_writer.addPage(page)  
...  
>>>
```

Not only is this loop slightly shorter than the loop in the first solution, but it doesn't rely on any prior knowledge of which pages need to be rotated. You could use a loop like this to rotate pages in any PDF without ever having to open it up and look at it.

To finish out the solution, write the contents of `pdf_writer` to a new file:

```
>>> with Path("ugly_rotated2.pdf").open(mode="wb") as output_file:  
...     pdf_writer.write(output_file)  
...  
>>>
```

Now you can open the `ugly_rotated2.pdf` file in your current working directory and compare it to the `ugly_rotated.pdf` file you generated earlier. They should look identical.

Important

One word of warning about the `/Rotate` key: it's not guaranteed to exist on a page.

If the `/Rotate` key doesn't exist, then that usually means that the page has not been rotated. However, that isn't always a safe assumption.

If a `PageObject` has no `/Rotate` key, then a `KeyError` will be raised when you try to access it. You can catch this exception with a `try ... except` block.

The value of `/Rotate` may not always be what you expect. For example, if you scan a paper document with the page rotated ninety degrees counterclockwise, then the contents of the PDF will appear rotated. However, the `/Rotate` key may have the value 0.

This is one of many quirks that can make working with PDF files frustrating. Sometimes you'll just need to open a PDF in a PDF reader program and manually figure things out.

Cropping Pages

Another common operation with PDFs is cropping pages. You might need to do this to split a single page into multiple pages or to extract just a small portion of a page, such as a signature or a figure.

For example, the chapter 14 `practice_files` folder includes a file called `half_and_half.pdf`. This PDF contains a portion of Hans Christian Andersen's *The Little Mermaid*.

Each page in this PDF has two columns. Let's split each page into two pages, one for each column.

To get started, import the `PdfFileReader` and `PdfFileWriter` classes from `PyPDF2` and the `Path` class from the `pathlib` module:

```
>>> from pathlib import Path  
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Now create a `Path` object for the `half_and_half.pdf` file:

```
>>> pdf_path = (  
...     Path.home() /  
...     "python-basics-exercises" /  
...     "ch14-interact-with-pdf-files" /  
...     "practice_files" /  
...     "half_and_half.pdf"  
... )
```

Next, create a new `PdfFileReader` object and get the first page of the PDF:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))  
>>> first_page = pdf_reader.getPage(0)
```

To crop the page, you first need to know a little bit more about how pages are structured. `PageObject` instances like `first_page` have a `.mediaBox` attribute that represents a rectangular area defining the boundaries of the page.

You can use IDLE's interactive window to explore the `.mediaBox` before using it to crop the page:

```
>>> first_page.mediaBox  
RectangleObject([0, 0, 792, 612])
```

The `.mediaBox` attribute returns a `RectangleObject`. This object is defined in the `PyPDF2` package and represents a rectangular area on the page.

The list `[0, 0, 792, 612]` in the output defines the rectangular area. The first two numbers are the x- and y-coordinates of the lower-left corner of the rectangle. The third and fourth numbers represent the width and height of the rectangle, respectively. The units of all of the values are points, which are equal to 1/72 of an inch.

`RectangleObject([0, 0, 792, 612])` represents a rectangular region with the lower-left corner at the origin, a width of 792 points, or 11 inches, and a height of 612 points, or 8.5 inches. Those are the dimensions of a standard letter-sized page in landscape orientation, which is used for the example PDF of *The Little Mermaid*. A letter-sized PDF page in portrait orientation would return the output `RectangleObject([0, 0, 612, 792])`.

A `RectangleObject` has four attributes that return the coordinates of the rectangle's corners: `.lowerLeft`, `.lowerRight`, `.upperLeft`, and `.upperRight`. Just like the width and height values, these coordinates are given in points.

You can use these four properties to get the coordinates of each corner of the `RectangleObject`:

```
>>> first_page.mediaBox.lowerLeft  
(0, 0)  
>>> first_page.mediaBox.lowerRight  
(792, 0)  
>>> first_page.mediaBox.upperLeft  
(0, 612)  
>>> first_page.mediaBox.upperRight  
(792, 612)
```

Each property returns a `tuple` containing the coordinates of the specified corner. You can access individual coordinates with square brackets just like you would any other Python tuple:

```
>>> first_page.mediaBox.upperRight[0]  
792  
>>> first_page.mediaBox.upperRight[1]  
612
```

You can alter the coordinates of a `mediaBox` by assigning a new tuple to one of its properties:

```
>>> first_page.mediaBox.upperLeft = (0, 480)
>>> first_page.mediaBox.upperLeft
(0, 480)
```

When you change the `.upperLeft` coordinates, the `.upperRight` attribute automatically adjusts to preserve a rectangular shape:

```
>>> first_page.mediaBox.upperRight
(792, 480)
```

When you alter the coordinates of the `RectangleObject` returned by `.mediaBox`, you effectively crop the page. The `first_page` object now contains only the information present within the boundaries of the new `RectangleObject`.

Go ahead and write the cropped page to a new PDF file:

```
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.addPage(first_page)
>>> with Path("cropped_page.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
>>>
```

If you open the `cropped_page.pdf` file in your current working directory, then you'll see that the top portion of the page has been removed.

How would you crop the page so that just the text on the left side of the page is visible? You would need to cut the horizontal dimensions of the page in half. You can achieve this by altering the `.upperRight` coordinates of the `.mediaBox` object. Let's see how that works.

The first thing you need to do is get new `PdfFileReader` and `PdfFileWriter` objects since you've just altered the first page in `pdf_reader` and then added it to `pdf_writer`:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
>>> pdf_writer = PdfFileWriter()
```

Now get the first page of the PDF:

```
>>> first_page = pdf_reader.getPage(0)
```

This time, let's work with a copy of the first page so that the page you just extracted stays intact. You can do that by importing the `copy` module from Python's standard library and using `deepcopy()` to make a copy of the page:

```
>>> import copy  
>>> left_side = copy.deepcopy(first_page)
```

Now you can alter `left_side` without changing any of the properties of `first_page`. That way, you can use `first_page` later to extract the text on the right side of the page.

Now you need to do a little bit of math. You already worked out that you need to move the upper right-hand corner of the `.mediaBox` to the top center of the page. To do that, you'll create a new tuple with the first component equal to half the original value and assign it to the `.upperRight` property.

First, get the current coordinates of the upper-right corner of the `.mediaBox`:

```
>>> current_coords = left_side.mediaBox.upperRight
```

Then create a new tuple whose first coordinate is half the value of the current coordinate and second coordinate is the same as the original:

```
>>> new_coords = (current_coords[0] / 2, current_coords[1])
```

Finally, assign the new coordinates to the `.upperRight` property:

```
>>> left_side.mediaBox.upperRight = new_coords
```

You've now cropped the original page to contain only the text on the left side! Let's extract the right side of the page next.

First get a new copy of `first_page`:

```
>>> right_side = copy.deepcopy(first_page)
```

Move the `.upperLeft` corner instead of the `.upperRight` corner:

```
>>> right_side.mediaBox.upperLeft = new_coords
```

This sets the upper-left corner to the same coordinates that you moved the upper-right corner to when extracting the left side of the page. So, `right_side.mediaBox` is now a rectangle whose upper-left corner is at the top center of the page and whose upper-right corner is at the top right of the page.

Finally, add the `left_side` and `right_side` pages to `pdf_writer` and write them to a new PDF file:

```
>>> pdf_writer.addPage(left_side)
>>> pdf_writer.addPage(right_side)
>>> with Path("cropped_pages.pdf").open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
...
>>>
```

Now open the `cropped_pages.pdf` file with a PDF reader. You should see a file with two pages, the first containing the text from the left-hand side of the original first page, and the second containing the text from the original right-hand side.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. The chapter 14 `practice_files` folder contains a PDF file called `split_and_rotate.pdf`. Create a new PDF in your home directory called `rotated.pdf` that contains the pages of `split_and_rotate.pdf` rotated counterclockwise 90 degrees.

- Using the `rotated.pdf` file you created in exercise 1, split each page of the PDF vertically in the middle. Create a new PDF in your home directory called `split.pdf` that contains all of the split pages.

`split.pdf` should have four pages with the numbers 1, 2, 3, and 4, in order.

14.6 Encrypting and Decrypting PDFs

Sometimes PDF files are password protected. With the `PyPDF2` package, you can work with encrypted PDF files as well as add password protection to existing PDFs.

Encrypting PDFs

You can add password protection to a PDF file using the `.encrypt()` method of a `PdfFileWriter()` instance. It has two main parameters:

- `user_pwd` sets the user password. This allows for opening and reading the PDF file.
- `owner_pwd` sets the owner password. This allows for opening the PDF without any restrictions, including editing.

Let's use `.encrypt()` to add a password to a PDF file. First, open the `newsletter.pdf` file in the chapter 14 `practice_files` directory:

```
>>> from pathlib import Path
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
>>> pdf_path = (
...     Path.home() /
...     "python-basics-exercises" /
...     "ch14-interact-with-pdf-files" /
...     "practice_files" /
...     "newsletter.pdf"
... )
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Now create a new `PdfFileWriter` instance and then add the pages from `pdf_reader` to it:

```
>>> pdf_writer = PdfFileWriter()
>>> pdf_writer.appendPagesFromReader(pdf_reader)
```

Next, add the password "SuperSecret" with `pdf_writer.encrypt()`:

```
>>> pdf_writer.encrypt(user_pwd="SuperSecret")
```

When you set only `user_pwd`, the `owner_pwd` argument defaults to the same string. So, the above line of code sets both the user and owner passwords.

Finally, write the encrypted PDF to an output file in your home directory called `newsletter_protected.pdf`:

```
>>> output_path = Path.home() / "newsletter_protected.pdf"
>>> with output_path.open(mode="wb") as output_file:
...     pdf_writer.write(output_file)
```

When you open the PDF with a PDF reader, you'll be prompted to enter a password. Enter "SuperSecret" to open the PDF.

If you need to set a separate owner password for the PDF, then pass a second string to the `owner_pwd` parameter:

```
>>> user_pwd = "SuperSecret"
>>> owner_pwd = "ReallySuperSecret"
>>> pdf_writer.encrypt(user_pwd=user_pwd, owner_pwd=owner_pwd)
```

In this example, the user password is "SuperSecret" and the owner password is "ReallySuperSecret".

When you encrypt a PDF file with a password and attempt to open it, you must provide the password before you can view its contents. This protection extends to reading from the PDF in a Python program. Next, let's see how to decrypt PDF files with PyPDF2.

Decrypting PDFs

To decrypt an encrypted PDF file, use the `.decrypt()` method of a `PdfFileReader` instance.

`.decrypt()` has a single parameter called `password` that you can use to provide the password for decryption. The privileges you have when opening the PDF depend on the argument you passed to the `password` parameter.

Let's open the encrypted `newsletter_protected.pdf` file that you created in the previous section and use `PyPDF2` to decrypt it.

First, create a new `PdfFileReader` instance with the path to the protected PDF:

```
>>> from pathlib import Path  
>>> from PyPDF2 import PdfFileReader, PdfFileWriter  
>>> pdf_path = Path.home() / "newsletter_protected.pdf"  
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Before you decrypt the PDF, check out what happens if you try to get the first page:

```
>>> pdf_reader.getPage(0)  
Traceback (most recent call last):  
  File "c:\realpython\venv\lib\site-packages\PyPDF2\pdf.py",  
    line 1617, in getObject  
      raise utils.PdfReadError("file has not been decrypted")  
PyPDF2.utils.PdfReadError: file has not been decrypted
```

A `PdfReadError` exception is raised, informing you that the PDF file has not been decrypted.

Note

The above traceback has been shortened to highlight the important parts. The traceback you see on your computer will be much longer.

In order to decrypt the file, you'll need to create a new a PdfFileReader instance:

```
>>> pdf_reader = PdfFileReader(str(pdf_path))
```

Go ahead and decrypt the file now:

```
>>> pdf_reader.decrypt(password="SuperSecret")  
1
```

.decrypt() returns an integer representing the success of the decryption:

- 0 indicates that the password is incorrect.
- 1 indicates that the user password was matched.
- 2 indicates that the owner password was matched.

Once you've decrypted the file, you can access the contents of the PDF:

```
>>> pdf_reader.getPage(0)  
{'/Contents': IndirectObject(7, 0), '/CropBox': [0, 0, 612, 792],  
 '/MediaBox': [0, 0, 612, 792], '/Parent': IndirectObject(1, 0),  
 '/Resources': IndirectObject(8, 0), '/Rotate': 0, '/Type': '/Page'}
```

Now you can extract text and crop or rotate pages to your heart's content!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. The chapter 14 practice_files folder contains a PDF file called top_secret.pdf. Using PdfFileWriter.encrypt(), encrypt the file with the user password `Unguessable`.

Save the encrypted file to your home directory with the file-name `top_secret_encrypted.pdf`.

2. Open the `top_secret_encrpyted.pdf` file you created in exercise 1, decrypt it, and print the text contained on the first page of the PDF.

14.7 Challenge: Unscramble a PDF

The chapter `14 practice_files` folder contains a PDF file called `scrambled.pdf` that has seven pages. Each page contains a number from 1 through 7, but they are out of order.

Additionally, some of the pages are rotated by 90, 180, or 270 degrees in either the clockwise or the counterclockwise direction.

Write a program that unscrambles the PDF by sorting the pages according to the number contained in the page text and rotates the page, if needed, so that it's upright.

Note

You can assume that every `PageObject` read from `scrambled.pdf` has a `"/Rotate"` key.

Save the unscrambled PDF to a file in your home directory called `unscrambled.pdf`.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

14.8 Creating a PDF File From Scratch

The `PYPDF2` package is great for reading and modifying existing PDF files, but it has a major limitation: you can't use it to create a new PDF file. In this section, you'll use the `ReportLab Toolkit` to generate PDF files from scratch.

ReportLab is a full-featured solution for creating PDFs. There is a commercial version that costs money to use, but a limited-feature open source version is also available.

Installing reportlab

To get started, you need to install `reportlab` with `pip`:

```
$ python3 -m pip install reportlab
```

You can verify the installation with `pip show`:

```
$ python3 -m pip show reportlab
Name: reportlab
Version: 3.5.34
Summary: The Reportlab Toolkit
Home-page: http://www.reportlab.com/
Author: Andy Robinson, Robin Becker, the ReportLab team
        and the community
Author-email: reportlab-users@lists2.reportlab.com
License: BSD license (see license.txt for details),
        Copyright (c) 2000-2018, ReportLab Inc.
Location: c:\realpython\venv\lib\site-packages
Requires: pillow
Required-by:
```

At the time of writing, the latest version of `reportlab` was 3.5.34. If you have IDLE open, then you'll need to restart it before you can use the `reportlab` package.

Using The `Canvas` Class

The main interface for creating PDFs with `reportlab` is the `Canvas` class, which is located in the `reportlab.pdfgen.canvas` module.

Open a new IDLE interactive window and type the following to import the `Canvas` class:

```
>>> from reportlab.pdfgen.canvas import Canvas
```

When you make a new `Canvas` instance, you need to provide a string with the filename of the PDF you're creating.

Go ahead and create a new `Canvas` instance for the file `hello.pdf`:

```
>>> canvas = Canvas("hello.pdf")
```

You now have a `Canvas` instance that you've assigned to the variable name `canvas` and that is associated with a file in your current working directory called `hello.pdf`. The file `hello.pdf` does not exist yet, though.

Let's add some text to the PDF. To do that, you use `.drawString()`:

```
>>> canvas.drawString(72, 72, "Hello, World")
```

The first two arguments passed to `.drawString()` determine the location on the canvas where the text is written. The first specifies the distance from the left edge of the canvas, and the second specifies the distance from the bottom edge.

The values passed to `.drawString()` are measured in points. Since a point equals $1/72$ of an inch, `.drawString(72, 72, "Hello, World")` draws the string "Hello, World" one inch from the left and one inch from the bottom of the page.

To save the PDF to a file, use `.save()`:

```
>>> canvas.save()
```

You now have a PDF file in your current working directory called `hello.pdf`. You can open it with a PDF reader and see the text `Hello, World` at the bottom of the page!

There are a few things to notice about the PDF you just created:

1. The default page size is A4, which is not the same as the standard US letter page size.
2. The font defaults to Helvetica with a font size of 12 points.

You're not stuck with these settings.

Setting the Page Size

When you instantiate a `Canvas` object, you can change the page size with the optional `pagesize` parameter. This parameter accepts a tuple of floating-point values representing the width and height of the page in points.

For example, to set the page size to 8.5 inches wide by 11 inches tall, you would create the following `Canvas`:

```
canvas = Canvas("hello.pdf", pagesize=(612.0, 792.0))
```

The tuple `(612.0, 792.0)` represents a letter-sized paper because 8.5 times 72 is 612, and 11 times 72 is 792.

If doing the math to convert points to inches or centimeters isn't your cup of tea, then you can use the `reportlab.lib.units` module to help you with the conversions. The `.units` module contains several helper objects, such as `inch` and `cm`, that you can use to simplify your conversions.

Go ahead and import the `inch` and `cm` objects from the `reportlab.lib.units` module:

```
>>> from reportlab.lib.units import inch, cm
```

Now you can inspect each object to see what they are:

```
>>> cm  
28.346456692913385  
>>> inch  
72.0
```

Both `cm` and `inch` are floating-point values. They represent the number of points contained in each unit. `cm` is 28.346456692913385 points, and `inch` is 72.0 points.

To use the units, multiply the unit name by the number of units that you want to convert to points. For example, here's how to use `inch` to set the page size to 8.5 inches wide by 11 inches tall:

```
>>> canvas = Canvas("hello.pdf", pagesize=(8.5 * inch, 11 * inch))
```

By passing a tuple to `pagesize`, you can create any size of page that you want. However, the `reportlab` package has some standard built-in page sizes that are easier to work with.

The page sizes are located in the `reportlab.lib.pagesizes` module. For example, to set the page size to letter, you can import the `LETTER` object from the `pagesizes` module and pass it to the `pagesize` parameter when instantiating your `Canvas`:

```
>>> from reportlab.lib.pagesizes import LETTER
>>> canvas = Canvas("hello.pdf", pagesize=LETTER)
```

If you inspect the `LETTER` object, then you'll see that it's a tuple of floats:

```
>>> LETTER
(612.0, 792.0)
```

The `reportlab.lib.pagesize` module contains many standard page sizes. Here are a few with their dimensions:

Page Size	Dimensions
A4	210 mm × 297 mm
LETTER	8.5 in × 11 in
LEGAL	8.5 in × 14 in
TABLOID	11 in × 17 in

In addition to these, the module contains definitions for all the [ISO 216](#) standard paper sizes.

Setting Font Properties

You can also change the font, font size, and font color when you write text to the canvas.

To change the font and font size, you can use `.setFont()`. First, create a new `Canvas` instance with the filename `font-example.pdf` and a letter page size:

```
>>> canvas = Canvas("font-example.pdf", pagesize=LETTER)
```

Then set the font to Times New Roman with a size of 18 points:

```
>>> canvas.setFont("Times-Roman", 18)
```

Finally, write the string "Times New Roman (18 pt)" to the canvas and save it:

```
>>> canvas.drawString(1 * inch, 10 * inch, "Times New Roman (18 pt)")  
>>> canvas.save()
```

With these settings, the text will be written one inch from the left side of the page and ten inches from the bottom. Open up the `font-example.pdf` file in your current working directory and check it out!

There are three fonts available by default:

1. "Courier"
2. "Helvetica"
3. "Times-Roman"

Each font has bolded and italicized variants. Here's a list of all the font variations available in `reportlab`:

- "Courier"
- "Courier-Bold"
- "Courier-BoldOblique"

- "Courier-Oblique"
- "Helvetica"
- "Helvetica-Bold"
- "Helvetica-BoldOblique"
- "Helvetica-Oblique"
- "Times-Bold"
- "Times-BoldItalic"
- "Times-Italic"
- "Times-Roman"

You can also set the font color using `.setFillColor()`. In the following example, you create a PDF file with blue text named `font-colors.pdf`:

```
from reportlab.lib.colors import blue
from reportlab.lib.pagesizes import LETTER
from reportlab.lib.units import inch
from reportlab.pdfgen.canvas import Canvas

canvas = Canvas("font-colors.pdf", pagesize=LETTER)

# Set font to Times New Roman with 12-point size
canvas.setFont("Times-Roman", 12)

# Draw blue text one inch from the left and ten
# inches from the bottom
canvas.setFillColor("blue")
canvas.drawString(1*inch, 10*inch, "Blue text")

# Save the PDF file
canvas.save()
```

`blue` is an object imported from the `reportlab.lib.colors` module. This module contains several common colors. A full list of colors can be found in the [reportlab source code](#).

The examples in this section highlight the basics of working with the `Canvas` object. But you've only scratched the surface. With `reportlab`, you can create tables, forms, and even high-quality graphics from scratch!

The [ReportLab User Guide](#) contains a plethora of examples of how to generate PDF documents from scratch. It's a great place to start if you're interested in learning more about creating PDFs with Python.

14.9 Summary and Additional Resources

In this chapter, you learned how to create and modify PDF files with the `PyPDF2` and `reportlab` packages.

With `PyPDF2`, you learned how to:

- Read PDF files and extract text using the `PdfFileReader` class
- Write new PDF files using the `PdfFileWriter` class
- Concatenate and merge PDF files using the `PdfFileMerger` class
- Rotate and crop PDF pages
- Encrypt and decrypt PDF files with passwords

You also had an introduction to creating PDF files from scratch with the `reportlab` package. You learned how to:

- Use the `Canvas` class
- Write text to a `Canvas` with `.drawString()`
- Set the font and font size with `.setFont()`
- Change the font color with `.setFillColor()`

`reportlab` is a powerful PDF creation tool, and you've only scratched the surface of what's possible.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-pdf

Additional Resources

To learn more about working with PDF files in Python, check out the following resources:

- “How to Work With a PDF in Python”
- ReportLab PDF Library User Guide

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 15

Working With Databases

In chapter 12, you learned how to store and retrieve data from files using Python. Another common way to store data is in a database.

A **database** is a structured system for storing data. It could be made up of several CSV files organized into directories or something more elaborate. A **database management system** is software that manages access to and interacts with a database.

Python comes with a lightweight database management system called SQLite that is perfect for learning how to work with databases.

In this chapter, you'll learn:

- How to create an SQLite database
- How to store and retrieve data from an SQLite database
- What packages are commonly used to work with other databases

Important

SQLite uses **structured query language (SQL)** to interact with its database. Some experience with SQL will be helpful when reading this chapter.

Let's dig in!

15.1 An Introduction to SQLite

There are numerous SQL database engines, and some are better suited to particular purposes than others. One of the simplest, most lightweight SQL database engines is [SQLite](#), which runs directly on your machine and comes bundled with the standard Python installation.

In this section, you'll learn how to use the `sqlite3` package to create a new SQLite database and store and retrieve data.

SQLite Basics

There are four basic steps to working with SQLite:

1. Import the `sqlite3` package.
2. Connect to an existing database or create a new database.
3. Execute SQL statements on the database.
4. Close the database connection.

Let's get started by exploring these four steps in IDLE's interactive window. Open IDLE and type the following:

```
>>> import sqlite3  
>>> connection = sqlite3.connect("test_database.db")
```

The `sqlite3.connect()` function is used to connect to or create a database.

When you execute `.connect("test_database.db")`, Python searches for an existing database called "test_database.db". If no database with that name is found, a new one is created in the current working directory.

To create a database in a different directory, you must specify the full path in the argument to `.connect()`.

Note

You can create an **in-memory database** by passing the string `":memory:"` to `.connect()`:

```
connection = sqlite3.connect(":memory:")
```

This is a good way to store data that only needs to exist while a program is running.

`.connect()` returns a `sqlite3.Connection` object. You can verify this with `type()`:

```
>>> type(connection)
<class 'sqlite3.Connection'>
```

The `Connection` object represents the connection between your program and the database. It has several attributes and methods that you can use to interact with the database.

To store and retrieve data, you need a `Cursor` object, which you can obtain with `connection.cursor()`:

```
>>> cursor = connection.cursor()
>>> type(cursor)
<class 'sqlite3.Cursor'>
```

The `sqlite3.Cursor` object is your gateway to interacting with the database. Using a `Cursor`, you can create database tables, execute SQL statements, and fetch query results.

Note

In database jargon, the term **cursor** refers to an object that is used to fetch results from a database query one row at a time.

Let's use the SQLite `datetime()` function to get the current local time:

```
>>> query = "SELECT datetime('now', 'localtime');"  
>>> results = cursor.execute(query)  
>>> results  
<sqlite3.Cursor object at 0x000001A27EB85E30>
```

"`SELECT datetime('now', 'localtime');`" is an SQL statement that returns the current date and time. You assign the query to the `query` variable and pass it to `cursor.execute()`. This runs the query against the database and returns a `Cursor` object, which you assign to the `results` variable.

You might be wondering where the time returned by `datetime()` is. To get the query results, use `results.fetchone()`, which returns a tuple containing the first row of results:

```
>>> row = results.fetchone()  
>>> row  
('2018-11-20 23:07:21',)
```

Since `.fetchone()` returns a tuple, you need to access the first element to get the string containing the date and time information:

```
>>> time = row[0]  
>>> time  
'2018-11-20 23:09:45'
```

Finally, call `connection.close()` to close the database connection:

```
>>> connection.close()
```

It's important to always close database connections when you're done using them to avoid leaving system resources hanging around after your program stops running.

Using `with` to Manage Your Database Connection

Recall from chapter 12 that you can use a `with` statement with `open()` to open a file and then automatically close it once the `with` block has executed. The same pattern applies to SQLite database connections and is the recommended way to open a database connection.

Here's the `datetime()` example from above using a `with` statement to manage the database connection:

```
>>> with sqlite3.connect("test_database.db") as connection:  
...     cursor = connection.cursor()  
...     query = "SELECT datetime('now', 'localtime');"  
...     results = cursor.execute(query)  
...     row = results.fetchone()  
...     time = row[0]  
...  
>>> time  
'2018-11-20 23:14:37'
```

In this example, you assign the `Connection` object returned by `sqlite3.connect()` to the `connection` variable in the `with` statement. The code in the `with` block creates a new `Cursor` object using `connection.cursor()` and then gets the current time with the `Cursor` object's `.execute()` and `.fetchone()` methods.

Managing your database connections in a `with` statement has many advantages. The resulting code is often cleaner and shorter than code written without a `with` statement. Moreover, as you'll see in the next example, any changes made to the database are saved automatically.

Working With Database Tables

You don't usually want to create a whole database just to get the current time. Databases are used to store and retrieve information. To store data in a database, you need to create a table and write some values to it.

Let's create a table called `People` with three columns: `FirstName`, `LastName`, and `Age`. The SQL query to create this table looks like this:

```
CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT);
```

Notice that `FirstName` and `LastName` are followed by the word `TEXT`, whereas `Age` is followed by the word `INT`. This tells SQLite that values in the `FirstName` and `LastName` columns are text values, while values in the `Age` column are integers.

Once you create the table, you can populate it with data using the `INSERT INTO` SQL command. The following query inserts the values `Ron`, `Obvious`, and `42` in the `FirstName`, `LastName`, and `Age` columns, respectively:

```
INSERT INTO People VALUES('Ron', 'Obvious', 42);
```

Note that the strings '`Ron`' and '`Obvious`' are delimited with single quotation marks. This still makes them valid Python strings, but more importantly, only strings delimited with single quotes are valid SQLite strings.

Important

When you write SQL queries as Python strings, you need to make sure that they're delimited with double quotation marks so that you can use single quotation marks inside them to delimit SQLite strings.

SQLite is not the only SQL database management system that follows the single quote convention. Keep an eye out for this whenever you work with any SQL database.

Let's walk through how to execute these statements and save the changes to the database. First, we'll do it without using a `with` statement.

In a new editor window, type in the following program:

```
import sqlite3

create_table = """
CREATE TABLE People(
    FirstName TEXT,
    LastName TEXT,
    Age INT
);"""

insert_values = """
INSERT INTO People VALUES(
    'Ron',
    'Obvious',
    42
);"""

connection = sqlite3.connect("test_database.db")
cursor = connection.cursor()
cursor.execute(create_table)
cursor.execute(insert_values)

connection.commit()
connection.close()
```

First, you create two strings containing SQL statements for creating the `People` table and inserting some data into it. You assign these strings to the `table` and `insert_values` variables.

You write both SQL statements using triple-quoted strings so you can format the SQL nicely. SQL ignores whitespace, so you can use spacing in the string to improve the readability of your Python code.

Then you create a `Connection` object with `sqlite3.connect()` and assign it to the `connection` variable. You also create a `cursor` object with `connection.cursor()` and use it to execute the two SQL statements.

Finally, you use `connection.commit()` to save the data to the database. **Commit** is database jargon for saving data. If you don't run `connection.commit()`, then the `People` table won't be created.

Save the file and press **F5** to run the program. `test_database.db` has a `People` table with one row in it. You can verify this in the interactive window:

```
>>> connection = sqlite3.connect("test_database.db")
>>> cursor = connection.cursor()
>>> query = "SELECT * FROM People;"
>>> results = cursor.execute(query)
>>> results.fetchone()
('Ron', 'Obvious', 42)
```

Now let's rewrite the program using a `with` statement to manage the database connection.

Before you can do anything, though, you need to delete the `People` table so that you can re-create it. Type the following code into the interactive window to remove the `People` table from the database:

```
>>> cursor.execute("DROP TABLE People;")
<sqlite3.Cursor object at 0x000001F739DB6650>
>>> connection.commit()
>>> connection.close()
```

Back in the editor window, change the program as follows:

```
import sqlite3

create_table = """
CREATE TABLE People(
    FirstName TEXT,
    LastName TEXT,
    Age INT
);"""
```

```
insert_values = """
INSERT INTO People VALUES(
    'Ron',
    'Obvious',
    42
);"""

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(create_table)
    cursor.execute(insert_values)
```

Neither `connection.close()` nor `connection.commit()` is needed. Any changes you make to the database will be automatically committed when the `with` block is done executing. This is another advantage of using a `with` statement to manage your database connection.

Executing Multiple SQL Statements

An **SQL script** is a collection of SQL statements separated by semi-colons (`;`) that can be run all at the same time. `Cursor` objects have an `.executescript()` method for executing SQL scripts.

The following program executes an SQL script that creates a `People` table and inserts some values into it:

```
import sqlite3

sql = """
DROP TABLE IF EXISTS People;
CREATE TABLE People(
    FirstName TEXT,
    LastName TEXT,
    Age INT
);
INSERT INTO People VALUES(
    'Ron',
```

```
'Obvious',
'42'
);"""
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.executescript(sql)
```

You can also execute many similar statements by using the `.executemany()` method and supplying a tuple of tuples in which each inner tuple supplies the information for a single command.

For instance, if you have a lot of people's information to insert into your `People` table, you can save this information in the following tuple of tuples:

```
people_values = (
    ("Ron", "Obvious", 42),
    ("Luigi", "Vercotti", 43),
    ("Arthur", "Belling", 28)
)
```

You can then insert all these people at once in a single line of code:

```
cursor.executemany("INSERT INTO People VALUES(?, ?, ?)", people_values)
```

Here, the question marks act as placeholders for the tuples contained in `people_values`. This is called a **parametrized statement**.

Each `?` represents a **parameter** that gets replaced by a value from `people_values` when the method is executed. The parameters are replaced in order. That is, the first `?` is replaced by the first value in `people_values`, the second `?` is replaced by the second value, and so on.

Avoid Security Issues With Parametrized Statements

For security reasons, especially when you need to interact with an SQL table based on user input, you should *always* use parametrized SQL statements. This is because the user could potentially supply a value that looks like SQL code and causes your SQL statement to behave in unexpected ways. This is called an **SQL injection** attack and, even if you aren't dealing with a **malicious user**, it can happen entirely by accident.

For example, suppose you want to insert a person into the `People` table based on user-supplied information. You might initially try something like the following:

```
import sqlite3

# Get person data from user
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))

# Execute insert statement for supplied person data
query = (
    "INSERT INTO People Values"
    f"('{first_name}', '{last_name}', {age});"
)
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(query)
```

What if the user's name includes an apostrophe? Try adding Flannery O'Connor to the table, and you'll see that she breaks the code. This is because the apostrophe gets mixed up with the single quotes in the line, making it appear that the SQL code ends earlier than you intend.

In this case, the code only causes an error, which is bad enough. In some cases, though, bad input can corrupt an entire table. Many other

hard-to-predict cases can break SQL tables and even delete portions of your database. To avoid this, you should always use parametrized statements.

The following code uses a parametrized statement to safely insert the user input into the database:

```
import sqlite3

first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))
data = (first_name, last_name, age)

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute("INSERT INTO People VALUES(?, ?, ?);", data)
```

Parametrization is also useful for updating a row in the database with an SQL UPDATE statement:

```
cursor.execute(
    "UPDATE People SET Age=? WHERE FirstName=? AND LastName=?;",
    (45, 'Luigi', 'Vercotti')
)
```

This code updates the value of the `Age` column to 45 for the row in which `FirstName` is set to 'Luigi' and `LastName` is set to 'Vercotti'.

Retrieving Data

Inserting and updating information in a database isn't all that helpful if you can't fetch that information from the database.

To fetch data from a database, you can use the `.fetchone()` and `.fetchall()` cursor methods. The `.fetchone()` method returns a single row from query results, while `.fetchall()` retrieves all the results of a query at once.

The following program illustrates how to use `.fetchall()`:

```
import sqlite3

values = (
    ("Ron", "Obvious", 42),
    ("Luigi", "Vercotti", 43),
    ("Arthur", "Belling", 28),
)

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute("DROP TABLE IF EXISTS People")
    cursor.execute("""
        CREATE TABLE People(
            FirstName TEXT,
            LastName TEXT,
            Age INT
        );""")
    cursor.executemany("INSERT INTO People VALUES(?, ?, ?);", values)

    # Select all first and last names from people over age 30
    cursor.execute(
        "SELECT FirstName, LastName FROM People WHERE Age > 30;"
    )
    for row in cursor.fetchall():
        print(row)
```

In the above program, you first drop the `People` table to destroy the changes made in the previous examples in this section. Then you re-create the `People` table and insert several values into it. Next, you `.execute()` a `SELECT` statement that returns the first and last names of all people over age 30.

Finally, `.fetchall()` returns the results of your query as a list of tuples in which each tuple contains a single row from the query results.

If you type the program into a new editor window and save and run the file, then you'll see the following output displayed in the interactive window:

```
('Ron', 'Obvious')
('Luigi', 'Vercotti')
```

Indeed, Ron and Luigi are the only people in the database who are over thirty years old.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create a new database with a table named `Roster` that has three fields: `Name`, `Species`, and `Age`. The `Name` and `Species` columns should be text fields, and the `Age` column should be an integer field.
2. Populate your new table with the following values:

Name	Species	Age
Benjamin Sisko	Human	40
Jadzia Dax	Trill	300
Kira Nerys	Bajoran	29

3. Update the `Name` of Jadzia Dax to be Ezri Dax.
4. Display the `Name` and `Age` of everyone in the table classified as Bajoran.

15.2 Libraries for Working With Other SQL Databases

If you have a particular type of SQL database that you'd like to access through Python, most of the basic syntax is likely to be identical to what you just learned for SQLite. However, you'll need to install an

additional package to interact with your database since SQLite is the only built-in option.

Many SQL variants and corresponding Python packages are available. Here are a few of the most commonly used and reliable open source alternatives to SQLite:

- [pyodbc](#) connects to ODBC (Open Database Connectivity) databases, such as Microsoft SQL Server.
- [psycopg2](#) connects to PostgreSQL databases.
- [PyMySQL](#) connects to MySQL databases.

One difference between SQLite and other database engines—besides the actual syntax of the SQL code, which changes slightly with most flavors of SQL—is that most database engines require a username and a password to connect. Check the documentation for the particular package you want to use to get the proper syntax for making a database connection.

The [SQLAlchemy](#) package is another popular option for working with databases. SQLAlchemy is an object-relational mapper, or ORM, that uses an object-oriented paradigm to build database queries. It can be configured to connect to a variety of databases. The object-oriented approach allows you to make queries without writing raw SQL statements.

15.3 Summary and Additional Resources

In this chapter, you learned how to interact with the SQLite database engine that comes with Python. SQLite is a small and light SQL database management system that can be used to store and retrieve data in your Python programs. To interact with SQLite in Python, you must import the `sqlite3` module.

To work with an SQLite database, you first need to connect to an existing database or create a new database with the `sqlite3.connect()` function, which returns a `Connection` object. Then you can use the `connection.cursor()` method to get a new `Cursor` object.

`Cursor` objects are used to execute SQL statements and retrieve query results. For example, `cursor.execute()` and `cursor.executescript()` are used to execute SQL queries. You can retrieve query results using the `cursor.fetchone()` and `cursor.fetchall()` methods.

Finally, you learned about several third-party packages that you can use to connect to other SQL databases, including `psycopg2`, which is used to connect to PostgreSQL databases, and `pyodbc` for Microsoft SQL Server. You also learned about the `SQLAlchemy` library, which provides a standard interface for connecting to a variety of SQL databases.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-databases

Additional Resources

Here are some more resources on working with databases:

- “[Introduction to Python SQL Libraries](#)”
- “[Preventing SQL Injection Attacks With Python](#)”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 16

Interacting With the Web

The Internet hosts perhaps the greatest source of information—and misinformation—on the planet.

Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from collecting and analyzing data from websites.

Web scraping is the process of collecting and parsing raw data from the Web, and the Python community has come up with some pretty powerful web scraping tools.

In this chapter, you'll learn how to:

- Parse website data using string methods and regular expressions
- Parse website data using an HTML parser
- Interact with forms and other website components

Important

Some experience with HTML will be helpful when reading this chapter.

Let's go!

16.1 Scrape and Parse Text From Websites

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones you'll create in this chapter. Websites do this for two possible reasons:

1. The site has a good reason to protect its data. For instance, Google Maps doesn't let you request too many results too quickly.
2. Making many repeated requests to a website's server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely.

Important

Before scraping a website's data, you should always check its acceptable use policy to see if accessing the website with automated tools is a violation of its terms of use. Legally, web scraping against the wishes of a website is very much a gray area.

*Please be aware that the following techniques **may be illegal** when used on websites that prohibit web scraping.*

Let's start by grabbing all the HTML code from a single web page. We'll use a page on Real Python that's been set up for use with this chapter.

Your First Web Scraper

Python's standard library has a package called `urllib` that contains tools for working with URLs. In particular, the `urllib.request` module contains a function called `urlopen()` that can be used to open a URL within a program.

In IDLE's interactive window, type the following to import `urlopen()`:

```
>>> from urllib.request import urlopen
```

The web page that we'll open is at the following URL:

```
>>> url = "http://olympus.realpython.org/profiles/aphrodite"
```

To open the web page, pass `url` to `urlopen()`:

```
>>> page = urlopen(url)
```

`urlopen()` returns an `HTTPResponse` object:

```
>>> page
<http.client.HTTPResponse object at 0x105fef820>
```

To extract the HTML from the page, first use the `HTTPResponse` object's `.read()` method, which returns a sequence of bytes. Then use `.decode()` to decode the bytes to a string using UTF-8:

```
>>> html_bytes = page.read()
>>> html = html_bytes.decode("utf-8")
```

Now you can print the HTML to see the contents of the web page:

```
>>> print(html)
<html>
<head>
<title>Profile: Aphrodite</title>
</head>
<body bgcolor="yellow">
<center>
<br><br>

<h2>Name: Aphrodite</h2>
<br><br>
Favorite animal: Dove
```

```
<br><br>
Favorite color: Red
<br><br>
Hometown: Mount Olympus
</center>
</body>
</html>
```

Once you have the HTML as text, you can extract information from it in a couple of different ways.

Extracting Text From HTML With String Methods

One way to extract information from a web page's HTML is to use string methods. For instance, you can use `.find()` to search through the text of the HTML for the `<title>` tags and extract the title of the web page.

Let's extract the title of the web page you requested in the previous example. If you know the index of the first character of the title and the first character of the closing `</title>` tag, then you can use a string slice to extract the title.

Since `.find()` returns the index of the first occurrence of a substring, you can get the index of the opening `<title>` tag by passing the string "`<title>`" to `.find()`:

```
>>> title_index = html.find("<title>")
>>> title_index
14
```

You don't want the index of the `<title>` tag, though. You want the index of the title itself. To get the index of the first letter in the title, you can add the length of the string "`<title>`" to `title_index`:

```
>>> start_index = title_index + len("<title>")
>>> start_index
21
```

Now get the index of the closing `</title>` tag by passing the string `"</title>"` to `.find()`:

```
>>> end_index = html.find("</title>")
>>> end_index
39
```

Finally, you can extract the title by slicing the `html` string:

```
>>> title = html[start_index:end_index]
>>> title
'Profile: Aphrodite'
```

Real-world HTML can be much more complicated and far less predictable than the HTML on the Aphrodite profile page. There's another profile page with some messier HTML that you can scrape at <http://olympus.realpython.org/profiles/poseidon>.

Try extracting the title from this new URL using the same method as the previous example:

```
>>> url = "http://olympus.realpython.org/profiles/poseidon"
>>> page = urlopen(url)
>>> html = page.read().decode("utf-8")
>>> start_index = html.find("<title>") + len("<title>")
>>> end_index = html.find("</title>")
>>> title = html[start_index:end_index]
>>> title
'\n<head>\n<title >Profile: Poseidon'
```

Whoops! There's a bit of HTML mixed in with the title. Why's that?

The HTML for the `/profiles/poseidon` page looks similar to the `/profiles/aphrodite` page, but there's a small difference. The opening `<title>` tag has an extra space before the closing angle bracket (`>`), rendering it as `<title >`.

`html.find("<title>")` returns `-1` because the exact substring "`<title>`" doesn't exist. When `-1` is added to `len("<title>")`, which is `7`, the `start_index` variable is assigned the value `6`.

The character at index `6` of the string `html` is a newline character (`\n`) right before the opening angle bracket (`<`) of the `<head>` tag. This means that `html[start_index:end_index]` returns all the HTML starting with that newline and ending just before the `</title>` tag.

These sorts of problems can occur in countless unpredictable ways. You need a more reliable way to extract text from HTML.

A Primer on Regular Expressions

Regular expressions—or **regexes** for short—are patterns that can be used to search for text within a string. Python supports regular expressions through the standard library's `re` module.

Note

Regular expressions aren't particular to Python. They're a general programming concept and can be used with any programming language.

To work with regular expressions, the first thing you need to do is import the `re` module:

```
import re
```

Regular expressions use special characters called **metacharacters** to denote different patterns. For instance, the asterisk character (`*`) stands for zero or more of whatever comes just before the asterisk.

In the following example, you use `.findall()` to find any text within a string that matches a given regular expression:

```
>>> re.findall("ab*c", "ac")  
['ac']
```

The first argument of `re.findall()` is the regular expression that you want to match, and the second argument is the string to test. In the above example, you search for the pattern "ab*c" in the string "ac".

The regular expression "ab*c" matches any part of the string that begins with an "a", ends with a "c", and has zero or more instances of "b" between the two. `re.findall()` returns a list of all matches. The string "ac" matches this pattern, so it's returned in the list.

Here's the same pattern applied to different strings:

```
>>> re.findall("ab*c", "abcd")
['abc']

>>> re.findall("ab*c", "acc")
['ac']

>>> re.findall("ab*c", "abcac")
['abc', 'ac']

>>> re.findall("ab*c", "abdc")
[]
```

Notice that if no match is found, then `findall()` returns an empty list.

Pattern matching is case sensitive. If you want to match this pattern regardless of the case, then you can pass a third argument with the value `re.IGNORECASE`:

```
>>> re.findall("ab*c", "ABC")
[]

>>> re.findall("ab*c", "ABC", re.IGNORECASE)
['ABC']
```

You can use a period (.) to stand for any single character in a regular expression. For instance, you could find all the strings that contain the letters "a" and "c" separated by a single character as follows:

```
>>> re.findall("a.c", "abc")
['abc']

>>> re.findall("a.c", "abbc")
[]

>>> re.findall("a.c", "ac")
[]

>>> re.findall("a.c", "acc")
['acc']
```

The pattern `.*` inside a regular expression stands for any character repeated any number of times. For instance, `"a.*c"` can be used to find every substring that starts with "a" and ends with "c", regardless of which letter—or letters—are in between:

```
>>> re.findall("a.*c", "abc")
['abc']

>>> re.findall("a.*c", "abbc")
['abbc']

>>> re.findall("a.*c", "ac")
['ac']

>>> re.findall("a.*c", "acc")
['acc']
```

Often, you use the `re.search()` function to search for a particular pattern inside a string. This function is somewhat more complicated than `re.findall()` because it returns an object called a `MatchObject` that stores different groups of data. This is because there might be matches inside other matches, and `re.search()` returns every possible result.

The details of the `MatchObject` are irrelevant here. For now, just know that calling `.group()` on a `MatchObject` will return the first and most inclusive result, which in most cases is just what you want:

```
>>> match_results = re.search("ab*c", "ABC", re.IGNORECASE)
>>> match_results.group()
'ABC'
```

There's one more function in the `re` module that's useful for parsing out text. `re.sub()`, which is short for *substitute*, allows you to replace text in a string that matches a regular expression with new text. It behaves sort of like the `.replace()` string method that you learned about in chapter 4.

The arguments passed to `re.sub()` are the regular expression, followed by the replacement text, followed by the string. Here's an example:

```
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*>", "ELEPHANTS", string)
>>> string
'Everything is ELEPHANTS.'
```

Perhaps that wasn't quite what you expected to happen.

`re.sub()` uses the regular expression "`<.*>`" to find and replace everything between the first `<` and last `>`, which spans from the beginning of `<replaced>` to the end of `<tags>`. This is because Python's regular expressions are **greedy**, meaning they try to find the longest possible match when characters like `*` are used.

Alternatively, you can use the non-greedy matching pattern `*?`, which works the same way as `*` except that it matches the shortest possible string of text:

```
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*?>", "ELEPHANTS", string)
>>> string
"Everything is ELEPHANTS if it's in ELEPHANTS."
```

Extracting Text From HTML With Regular Expressions

Armed with all this knowledge, let's now try to parse out the title from <http://olympus.realpython.org/profiles/dionysus>, which includes this rather carelessly written line of HTML:

```
<TITLE>Profile: Dionysus</TITLE>
```

The `.find()` method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code quickly and efficiently:

```
import re
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")

pattern = "<title.*?>.*?</title.*?>"
match_results = re.search(pattern, html, re.IGNORECASE)
title = match_results.group()
title = re.sub("<.*?>", "", title) # Remove HTML tags

print(title)
```

Let's take a closer look at the first regular expression in the `pattern` string by breaking it down into three parts:

1. `<title.*?>` matches the opening `<TITLE>` tag in `html`. The `<title` part of the pattern matches with `<TITLE` because `re.search()` is called with `re.IGNORECASE`, and `.*?>` matches any text after `<TITLE` up to the first instance of `>`.
2. `.*?` non-greedily matches all text after the opening `<TITLE>`, stopping at the first match for `</title.*?>`.

3. `</title.*?>` differs from the first pattern only in its use of the `/` character, so it matches the closing `</title />` tag in `html`.

The second regular expression, the string "`<.*?>`", also uses the non-greedy `.*?` to match all the HTML tags in the `title` string. By replacing any matches with "", `re.sub()` removes all the tags and returns only the text.

Regular expressions are a powerful tool when used correctly. This introduction barely scratches the surface. For more about regular expressions and how to use them, check out *Real Python*'s two-part series “[Regular Expressions: Regexes in Python](#).”

Note

Web scraping can be tedious. No two websites are organized the same way, and HTML is often messy. Moreover, websites change over time. Web scrapers that work today are not guaranteed to work next year—or next week, for that matter!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that grabs the full HTML from the web page at <http://olympus.realpython.org/profiles/dionysus>.
2. Use the string `.find()` method to display the text following “Name:” and “Favorite Color:” (not including any leading spaces or trailing HTML tags that might appear on the same line).
3. Repeat the previous exercise using regular expressions. The end of each pattern should be a "`<`" (the start of an HTML tag) or a new-line character, and you should remove any extra spaces or newline characters from the resulting text using the string `.strip()` method.

16.2 Use an HTML Parser to Scrape Websites

Although regular expressions are great for pattern matching in general, sometimes it's easier to use an HTML parser that's explicitly designed for parsing out HTML pages. There are many Python tools written for this purpose, but the [Beautiful Soup](#) library is a good one to start with.

Install BeautifulSoup

To install BeautifulSoup, you can run the following in your terminal:

```
$ python3 -m pip install beautifulsoup4
```

Run `pip show` to see the details of the package you just installed:

```
$ python3 -m pip show beautifulsoup4
Name: beautifulsoup4
Version: 4.9.1
Summary: Screen-scraping library
Home-page: http://www.crummy.com/software/BeautifulSoup/bs4/
Author: Leonard Richardson
Author-email: leonardr@segfault.org
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

In particular, notice that the latest version at the time of writing was 4.9.1.

Create a BeautifulSoup Object

Type the following program into a new editor window:

```
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")
soup = BeautifulSoup(html, "html.parser")
```

This program does three things:

1. Opens the URL `http://olympus.realpython.org/profiles/dionysus` using `urlopen()` from the `urllib.request` module
2. Reads the HTML from the page as a string and assigns it to the `html` variable
3. Creates a `BeautifulSoup` object and assigns it to the `soup` variable

The `BeautifulSoup` object assigned to `soup` is created with two arguments. The first argument is the HTML to be parsed, and the second argument, the string `"html.parser"`, tells the object which parser to use behind the scenes. `"html.parser"` represents Python's built-in HTML parser.

Use a BeautifulSoup Object

Save and run the above program. When it's finished running, you can use the `soup` variable in the interactive window to parse the content of `html` in various ways.

For example, `BeautifulSoup` objects have a `.get_text()` method that can be used to extract all the text from the document and automatically remove any HTML tags.

Type the following code into IDLE's interactive window:

```
>>> print(soup.get_text())
```

Profile: Dionysus

Name: Dionysus

Hometown: Mount Olympus

Favorite animal: Leopard

Favorite Color: Wine

There are a lot of blank lines in this output! These are the result of newline characters in the HTML document's text. You can remove them with the string `.replace()` method if you need to.

Often, you need to get only specific text from an HTML document. Using Beautiful Soup first to extract the text and then using the `.find()` string method is *sometimes* easier than working with regular expressions.

However, sometimes the HTML tags themselves are the elements that point out the data you want to retrieve. For instance, perhaps you want to retrieve the URLs for all the images on the page. These links are contained in the `src` attribute of `` HTML tags.

In this case, you can use `find_all()` to return a list of all instances of that particular tag:

```
>>> soup.find_all("img")
[, ]
```

This returns a list of all `` tags in the HTML document. The objects in the list look like they might be strings representing the tags, but they're actually instances of the `Tag` object provided by Beautiful Soup. `Tag` objects provide a simple interface for working with the information they contain.

Let's explore this a little by first unpacking the `Tag` objects from the list:

```
>>> image1, image2 = soup.find_all("img")
```

Each `Tag` object has a `.name` property that returns a string containing the HTML tag type:

```
>>> image1.name  
'img'
```

You can access the HTML attributes of the `Tag` object by putting their name between square brackets, just as if the attributes were keys in a dictionary.

For example, the `` tag has a single attribute, `src`, with the value `"/static/dionysus.jpg"`. Likewise, an HTML tag such as the link `` has two attributes, `href` and `target`.

To get the source of the images in the Dionysus profile page, you access the `src` attribute using the dictionary notation mentioned above:

```
>>> image1["src"]  
'/static/dionysus.jpg'  
  
>>> image2["src"]  
'/static/grapes.png'
```

Certain tags in HTML documents can be accessed by properties of the `Tag` object. For example, to get the `<title>` tag in a document, you can use the `.title` property:

```
>>> soup.title  
<title>Profile: Dionysus</title>
```

If you look at the source of the Dionysus profile by navigating to the URL <http://olympus.realpython.org/profiles/dionysus>, right-clicking on the page, and selecting *View page source*, then you'll notice that the `<title>` tag as written in the document looks like this:

```
<title >Profile: Dionysus</title/>
```

Beautiful Soup automatically cleans up the tags for you by removing the extra space in the opening tag and the extraneous forward slash (/) in the closing tag.

You can also retrieve just the string between the title tags with the `.string` property of the `Tag` object:

```
>>> soup.title.string  
'Profile: Dionysus'
```

One of the more useful features of Beautiful Soup is the ability to search for specific kinds of tags whose attributes match certain values. For example, if you want to find all the `` tags that have a `src` attribute equal to the value `/static/dionysus.jpg`, then you can provide the following additional argument to `.find_all()`:

```
>>> soup.find_all("img", src="/static/dionysus.jpg")  
[]
```

This example is somewhat arbitrary, and the usefulness of this technique may not be immediately apparent. If you spend some time browsing websites and viewing their page sources, then you'll notice that many websites have extremely complicated HTML structures.

When scraping data from websites, you're often interested in particular parts of the page. By spending some time looking through the HTML document, you can identify tags with unique attributes that you can use to extract the data you need.

Then, instead of relying on complicated regular expressions or using `.find()` to search through the document, you can directly access the particular tag you're interested in and extract the data you need.

In some cases, you may find that Beautiful Soup doesn't offer the functionality you need. The `lxml` library is somewhat trickier to get started with but offers far more flexibility than Beautiful Soup for parsing HTML documents. You may want to check it out once you're comfortable using Beautiful Soup.

Note

HTML parsers like Beautiful Soup can save you a lot of time and effort when it comes to locating specific data in web pages. However, sometimes HTML is so poorly written and disorganized that even a sophisticated parser like Beautiful Soup can't interpret the HTML tags properly.

In this case, you're often left to your own devices (namely, `.find()` and regex) to try to parse out the information you need.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Write a program that grabs the full HTML from the web page at <http://olympus.realpython.org/profiles>.
2. Using Beautiful Soup, parse out a list of all the links on the page by looking for HTML tags with the name `a` and retrieving the value taken on by the `href` attribute of each tag.
3. Get the HTML from each of the pages in the list by adding the full path to the filename, and display the text (without HTML tags) on each page using Beautiful Soup's `.get_text()` method.

16.3 Interact With HTML Forms

The `urllib` module you've been working with so far in this chapter is well suited for requesting the contents of a web page. Sometimes, though, you need to interact with a web page to obtain the content you need. For example, you might need to submit a form or click a button to display hidden content.

The Python standard library doesn't provide a built-in means for working with web pages interactively, but many third-party packages are available from PyPI. Among these, [MechanicalSoup](#) is a popular and relatively straightforward package to use.

In essence, MechanicalSoup installs what's known as a **headless browser**, which is a web browser with no graphical user interface. This browser is controlled programmatically via a Python program.

Install MechanicalSoup

You can install MechanicalSoup with `pip` in your terminal:

```
$ python3 -m pip install MechanicalSoup
```

You can now view some details about the package with `pip show`:

```
$ python3 -m pip show mechanicalsoup
Name: MechanicalSoup
Version: 0.12.0
Summary: A Python library for automating interaction with websites
Home-page: https://mechanicalsoup.readthedocs.io/
Author: UNKNOWN
Author-email: UNKNOWN
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires: requests, beautifulsoup4, six, lxml
Required-by:
```

In particular, notice that the latest version at the time of writing was 0.12.0. You'll need to close and restart your IDLE session for MechanicalSoup to load and be recognized after it's been installed.

Create a Browser Object

Type the following into IDLE's interactive window:

```
>>> import mechanicalsoup  
>>> browser = mechanicalsoup.Browser()
```

Browser objects represent the headless web browser. You can use them to request a page from the Internet by passing a URL to their `.get()` method:

```
>>> url = "http://olympus.realpython.org/login"  
>>> page = browser.get(url)
```

`page` is a `Response` object that stores the response from requesting the URL from the browser:

```
>>> page  
<Response [200]>
```

The number 200 represents the **status code** returned by the request. A status code of 200 means that the request was successful. An unsuccessful request might show a status code of 404 if the URL doesn't exist or a status code of 500 if there was a server error when making the request.

MechanicalSoup uses BeautifulSoup to parse the HTML from the request. `page` has a `.soup` attribute that represents a `BeautifulSoup` object:

```
>>> type(page.soup)  
<class 'bs4.BeautifulSoup'>
```

You can view the HTML by inspecting the `.soup` attribute:

```
>>> page.soup
<html>
<head>
<title>Log In</title>
</head>
<body bgcolor="yellow">
<center>
<br/><br/>
<h2>Please log in to access Mount Olympus:</h2>
<br/><br/>
<form action="/login" method="post" name="login">
Username: <input name="user" type="text"/><br/>
Password: <input name="pwd" type="password"/><br/><br/>
<input type="submit" value="Submit"/>
</form>
</center>
</body>
</html>
```

Notice this page has a `<form>` on it with `<input>` elements for a username and a password.

Submitting a Form With MechanicalSoup

Open <http://olympus.realpython.org/login> in a browser and look at it yourself before moving on. Try typing in a random username and password combination. If you guess incorrectly, then the message “Wrong username or password!” is displayed at the bottom of the page.

However, if you provide the correct login credentials (username `zeus` and password `ThunderDude`), then you’re redirected to the `/profiles` page.

In the next example, you’ll see how to use MechanicalSoup to fill out and submit this form using Python!

The important section of HTML code is the login form—that is, everything inside the `<form>` tags. The `<form>` on this page has the `name` attribute set to `login`. This form contains two `<input>` elements, one named `user` and the other named `pwd`. The third `<input>` element is the Submit button.

Now that you know the underlying structure of the login form, as well as the credentials needed to log in, let's take a look at a program that fills the form out and submits it.

In a new editor window, type in the following program:

```
import mechanize

# 1
browser = mechanize.Browser()
url = "http://olympus.realpython.org/login"
login_page = browser.get(url)
login_html = login_page.read()

# 2
form = login_html.select("form")[0]
form.select("input")[0]["value"] = "zeus"
form.select("input")[1]["value"] = "ThunderDude"

# 3
profiles_page = browser.submit(form, login_page.url)
```

Save the file and press `F5` to run it. You can confirm that you successfully logged in by typing the following into the interactive window:

```
>>> profiles_page.url
'http://olympus.realpython.org/profiles'
```

Let's break down the above example:

1. You create a `Browser` instance and use it to request the `http://olympus.realpython.org/login` page. You assign the

HTML content of the page to the `login_html` variable using the `.soup` property.

2. `login_html.select("form")` returns a list of all `<form>` elements on the page. Since the page has only one `<form>` element, you can access the form by retrieving the element at index 0 of the list. The next two lines select the username and password inputs and set their value to "zeus" and "ThunderDude", respectively.
3. You submit the form with `browser.submit()`. Notice that you pass two arguments to this method, the `form` object and the URL of the `login_page`, which you access via `login_page.url`.

In the interactive window, you confirm that the submission successfully redirected to the `/profiles` page. If something had gone wrong, then the value of `profiles_page.url` would still be "`http://olympus.realpython.org/login`".

Note

Hackers can use automated programs like the one above to **brute force** logins by rapidly trying many different usernames and passwords until they find a working combination.

Besides this being highly illegal, almost all websites these days lock you out and report your IP address if they see you making too many failed requests, so don't try it!

Now that we have the `profiles_page` variable set, let's see how to programmatically obtain the URL for each link on the `/profiles` page.

To do this, you use `.select()` again, this time passing the string "a" to select all the `<a>` anchor elements on the page:

```
>>> links = profiles_page_soup.select("a")
```

Now you can iterate over each link and print the `href` attribute:

```
>>> for link in links:  
...     address = link["href"]  
...     text = link.text  
...     print(f"{text}: {address}")  
  
Aphrodite: /profiles/aphrodite  
Poseidon: /profiles/poseidon  
Dionysus: /profiles/dionysus
```

The URLs contained in each `href` attribute are relative URLs, which aren't very helpful if you want to navigate to them later using `MechanicalSoup`.

If you happen to know the full URL, then you can assign the portion needed to construct a full URL. In this case, the base URL is just `http://olympus.realpython.org`. Then you can concatenate the base URL with the relative URLs found in the `href` attribute:

```
>>> base_url = "http://olympus.realpython.org"  
>>> for link in links:  
...     address = base_url + link["href"]  
...     text = link.text  
...     print(f"{text}: {address}")  
  
Aphrodite: http://olympus.realpython.org/profiles/aphrodite  
Poseidon: http://olympus.realpython.org/profiles/poseidon  
Dionysus: http://olympus.realpython.org/profiles/dionysus
```

You can do a lot with just `.get()`, `.select()`, and `.submit()`. That said, `MechanicalSoup` is capable of much more. To learn more about `MechanicalSoup`, check out the [official docs](#).

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Use `MechanicalSoup` to provide the correct username ("zeus")

and password ("ThunderDude") to the login page submission form located at <http://olympus.realpython.org/login>.

2. Display the title of the current page to determine that you've been redirected to the [/profiles](#) page.
3. Use MechanicalSoup to return to the [login page](#) by going back to the previous page.
4. Provide an incorrect username and password to the login form, then search the HTML of the returned web page for the text "Wrong username or password!" to determine that the login process failed.

16.4 Interact With Websites in Real Time

Sometimes you want to be able to fetch real-time data from a website that offers continually updated information.

In the dark days before you learned Python programming, you had to sit in front of a browser, repeatedly refreshing to page to see if updated content was available. But now you can automate this process using the `.get()` method of the MechanicalSoup `Browser` object.

Open a browser and navigate to <http://olympus.realpython.org/dice>. This page simulates a roll of a six-sided die, updating the result each time you refresh the browser. You'll write a program that repeatedly scrapes this page for a new result.

The first thing you need to do is determine which element on the page contains the result of the die roll. Do this now by right-clicking anywhere on the page and selecting *View page source*. A little more than halfway down the HTML code is an `<h2>` tag that looks like this:

```
<h2 id="result">4</h2>
```

The text of the `<h2>` tag might be different for you, but this is the page element you need for scraping the result.

Note

For this example, you can easily check that there's only one element on the page with `id="result"`. Although the `id` attribute is supposed to be unique, in practice you should always check that the element you're interested in is uniquely identified.

Let's start by writing a simple program that opens the [/dice](#) page, scrapes the result, and prints it to the console:

```
import mechanize

browser = mechanize.Browser()
page = browser.get("http://olympus.realpython.org/dice")
tag = page.soup.select("#result")[0]
result = tag.text

print(f"The result of your dice roll is: {result}")
```

This example uses the `BeautifulSoup` object's `.select()` method to find the element with `id=result`. The string "`#result`" that you pass to `.select()` uses the [CSS ID selector](#) `#` to indicate that `result` is an `id` value.

To periodically get a new result, you'll need to create a loop that loads the page at each step. So everything below the line `browser = mechanize.Browser()` in the above code needs to go in the body of the loop.

For this example, let's get four rolls of the dice at ten-second intervals. To do that, the last line of your code needs to tell Python to pause running for ten seconds. You can do this with `sleep()` from Python's `time` module. `sleep()` takes a single argument that represents the amount of time to sleep in seconds.

Here's an example that illustrates how `sleep()` works:

```
import time

print("I'm about to wait for five seconds...")
time.sleep(5)
print("Done waiting!")
```

When you run this code, you'll see that the "Done waiting!" message isn't displayed until 5 seconds have passed from when the first `print()` function was executed.

For the die roll example, you'll need to pass the number `10` to `sleep()`. Here's the updated program:

```
import time
import mechanicalsoup

browser = mechanicalsoup.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.soup.select("#result")[0]
    result = tag.text
    print(f"The result of your dice roll is: {result}")
    time.sleep(10)
```

When you run the program, you'll immediately see the first result printed to the console. After ten seconds, the second result is displayed, then the third, and finally the fourth. What happens after the fourth result is printed?

The program continues running for another ten seconds before it finally stops!

Well, of course it does—that's what you told it to do! But it's kind of a waste of time. You can stop it from doing this by using an `if` statement to run `time.sleep()` for only the first three requests:

```
import time
import mechanicalsoup

browser = mechanicalsoup.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.soup.select("#result")[0]
    result = tag.text
    print(f"The result of your dice roll is: {result}")

    # Wait 10 seconds if this isn't the last request
    if i < 3:
        time.sleep(10)
```

With techniques like this, you can scrape data from websites that periodically update their data. However, you should be aware that requesting a page multiple times in rapid succession can be seen as suspicious, or even malicious, use of a website.

Important

Most websites publish a Terms of Use document. You can often find a link to it in the website's footer.

Always read this document before attempting to scrape data from a website. If you can't find the Terms of Use, try to contact the website owner and ask them if they have any policies regarding request volume.

Failure to comply with the Terms of Use could result in your IP being blocked, so be careful and be respectful!

It's even possible to crash a server with an excessive number of requests, so you can imagine that many websites are concerned about the volume of requests to their server! Always check the Terms of Use and be respectful when sending multiple requests to a website.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Repeat the example in this section to scrape the die roll result, but also include the current time of the quote as obtained from the web page. This time can be taken from part of a string inside a `<p>` tag that appears shortly after the result of the roll in the web page's HTML.

16.5 Summary and Additional Resources

Although it's possible to parse data from the Web using tools in Python's standard library, there are many tools on PyPI that can help simplify the process.

In this chapter, you learned how to:

- Request a web page using Python's built-in `urllib` module
- Parse HTML using BeautifulSoup
- Interact with web forms using MechanicalSoup
- Repeatedly request data from a website to check for updates

Writing automated web scraping programs is fun, and the Internet has no shortage of content that can lead to all sorts of exciting projects.

Just remember, not everyone wants you pulling data from their web servers. Always check a website's Terms of Use before you start scraping, and be respectful about how you time your web requests so that you don't flood a server with traffic.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-web

Additional Resources

For more information on interacting with the Web with Python, check out the following resources:

- “Beautiful Soup: Build a Web Scraper With Python”
- “API Integration in Python”

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 17

Scientific Computing and Graphing

Python is one of the leading programming languages in scientific computing and data science.

Python's popularity in this area is due, in part, to the wealth of third-party packages available on PyPI for manipulating and visualizing data.

From working with large arrays of data to visualizing data in plots and charts, Python's ecosystem has the tools you need.

In this chapter, you'll learn how to:

- Work with arrays of data using NumPy
- Create charts and plots with Matplotlib

Note

Some knowledge of matrices is assumed in this chapter. If you're unfamiliar with matrices or have no interest in scientific computing, then you may safely skip this chapter.

Let's dive in!

17.1 Use NumPy for Matrix Manipulation

In this section, you'll learn how to store and manipulate matrices of data using the [NumPy](#) package. Before getting to that, though, let's take a look at the problem NumPy solves.

If you've ever taken a course in linear algebra, then you may recall that a matrix is a rectangular array of numbers. You can create a matrix in pure Python with a list of lists:

```
>>> matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

This seemingly works well. You can access individual elements of the matrix using their indices. For example, here's how you would access the second element of the first row of the matrix:

```
>>> matrix[0][1]
2
```

Now suppose you want to multiply every element of the matrix by 2. To do this, you need to write a nested `for` loop that loops over every element in each row of the matrix, like this:

```
>>> for row in matrix:
...     for i in range(len(row)):
...         row[i] = row[i] * 2
...
>>> matrix
[[2, 4, 6], [8, 10, 12], [14, 16, 18]]
```

While this may not seem so hard, the point is that in pure Python, you need to do a lot of work from scratch to implement even simple linear algebra tasks.

For working with multidimensional arrays, NumPy provides nearly all the functionality you need out of the box and is more efficient than pure Python. NumPy is written in the C language and uses sophisticated algorithms for efficient computation.

Note

NumPy is useful for more than just scientific computing. For instance, perhaps you’re designing a game and need an easy way to manipulate a grid of values with rows and columns. NumPy arrays are a great way to store two-dimensional data.

Install NumPy

Before you can work with NumPy, you’ll need to install it using pip:

```
$ python3 -m pip install numpy
```

Once NumPy has finished installing, you can see some details about the package by running `pip show`:

```
$ python3 -m pip show numpy
Name: numpy
Version: 1.18.5
Summary: NumPy: array processing for numbers, strings,
         records, and objects.
Home-page: http://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

In particular, notice that the latest version at the time of writing was version 1.18.5.

Create a NumPy array

Now that you have NumPy installed, let's create the same matrix from the first example in this section. Matrices in NumPy are instances of the `ndarray` object, which stands for ***n-dimensional array***.

Note

An *n*-dimensional array is an array with *n* dimensions. For example, a one-dimensional array is a list. A two-dimensional array is a matrix. Arrays can also have three, four, or more dimensions.

In this section, we'll focus on arrays with one or two dimensions.

To create an `ndarray` object, you can use the `array` alias. You initialize array objects with a list of lists, so to re-create the matrix from the first example as a NumPy array, you can do the following:

```
>>> import numpy as np  
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
>>> matrix  
array([[1, 2, 3],  
       [4, 5, 6],  
       [7, 8, 9]])
```

Notice how NumPy displays the matrix in a conveniently readable format. This is even true when you print the matrix with `print()`:

```
>>> print(matrix)  
[[1 2 3]  
 [4 5 6]  
 [7 8 9]]
```

Accessing individual elements of the array works just like accessing elements in a list of lists:

```
>>> matrix[0][1]  
2
```

You can optionally access elements with just a single set of square brackets by separating the indices with a comma:

```
>>> matrix[0, 1]  
2
```

At this point, you might be wondering what the major difference is between a NumPy array and a Python list. For starters, NumPy arrays can hold only objects of the same type, like the all-number matrix above, whereas Python lists can hold objects of mixed types.

Check out what happens if you try to create an array with mixed types:

```
>>> np.array([[1, 2, 3], ["a", "b", "c"]])  
array([[1, 2, 3],  
       ['a', 'b', 'c']], dtype='<U11')
```

NumPy doesn't raise an error. Instead, it converts every element to a string. The `dtype='<U11'` that you see in the above output means that this array can store only Unicode strings whose length is at most 11 bytes.

Automatic data type conversions can be helpful, but they're also a potential source of frustration. Data types may not get converted in the manner you expect.

It's generally a good idea to handle any type conversion *before* initializing an array object. That way you can be sure that the data type stored in your array matches your expectations.

In NumPy, each dimension in an array is called an **axis**. The previous matrices you've seen have two axes. Arrays with two axes are called **two-dimensional arrays**.

Here's an example of a **three-dimensional array**:

```
>>> matrix = np.array([
...     [[1, 2, 3], [4, 5, 6]],
...     [[7, 8, 9], [10, 11, 12]],
...     [[13, 14, 15], [16, 17, 18]]
... ])
```

To access an element of the above array, you need to supply three indices:

```
>>> matrix[0][1][2]
6
>>> matrix[0, 1, 2]
6
```

If you think creating the above three-dimensional array looks confusing, you'll see a better way to create higher-dimensional arrays later in this section.

Array Operations

Once you have an `array` object created, you can start to unleash the power of NumPy and perform some operations.

Recall from the earlier example how you had to write a nested `for` loop to multiply each element in a matrix by 2. In NumPy, this operation is as simple as multiplying your `array` object by 2:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> 2 * A
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

Operations between two matrices are performed **element-wise**, so that the operator is applied to corresponding elements in the matrix:

```
>>> B = np.array([[5, 4, 3], [7, 6, 5], [9, 8, 7]])
>>> C = B - A
>>> C
array([[ 4,  2,  0],
       [ 3,  1, -1],
       [ 2,  0, -2]])
```

Notice how $C[0][0]$ is $B[0][0] - A[0][0]$. The same is true for every pair of indices. All of the basic arithmetic operators ($+$, $-$, $*$, $/$) operate on arrays element-wise.

For example, multiplying two arrays with the `*` operator does *not* compute the product of two matrices:

```
>>> A = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
>>> A * A
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

To calculate an actual [matrix product](#), you can use the `@` operator:

```
>>> A @ A
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

The `@` operator was introduced in Python 3.5, so if you're using an older version of Python, then you'll need to multiply matrices differently. NumPy provides a function called `matmul()` for multiplying two matrices:

```
>>> np.matmul(matrix, matrix)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

The @ operator relies on the `np.matmul()` function internally, so there's no real difference between the two methods.

Other common array operations are listed here:

```
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])  
  
>>> # Get a tuple of axis length  
>>> matrix.shape  
(3, 3)  
  
>>> # Get an array of the diagonal entries  
>>> matrix.diagonal()  
array([1, 5, 9])  
  
>>> # Get a one-dimensional array of all entries  
>>> matrix.flatten()  
array([1, 2, 3, 4, 5, 6, 7, 8, 9])  
  
>>> # Get the transpose of an array  
>>> matrix.transpose()  
array([[1, 4, 7],  
       [2, 5, 8],  
       [3, 6, 9]])  
  
>>> # Calculate the minimum entry  
>>> matrix.min()  
1  
  
>>> # Calculate the maximum entry  
>>> matrix.max()  
9  
  
>>> # Calculate the average value of all entries  
>>> matrix.mean()  
5.0
```

```
>>> # Calculate the sum of all entries
>>> matrix.sum()
45
```

Now let's look at some ways to create new arrays from old ones.

Stacking and Shaping Arrays

If their axis sizes match, two arrays can be stacked vertically using `np.vstack()` or horizontally using `np.hstack()`:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6]])
>>> B = np.array([[7, 8, 9], [10, 11, 12]])

>>> np.vstack([A, B])
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12]])

>>> np.hstack([A, B])
array([[ 1,  2,  3,  7,  8,  9],
       [ 4,  5,  6, 10, 11, 12]])
```

You can also reshape arrays with `np.reshape()`:

```
>>> A.reshape(6, 1)
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6]])
```

Note that `.reshape()` returns a new array and doesn't modify the original array in place.

The total size of the reshaped array must match the size of the original array. For instance, you can't execute `matrix.reshape(2, 5)`:

```
>>> A.reshape(2, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 6 into shape (2, 5)
```

In this case, you're trying to shape an array with nine entries into an array with two columns and five rows. This requires a total of ten entries.

`np.reshape()` can be particularly helpful in combination with `np.arange()`, which is NumPy's equivalent to Python's `range()` function. The main difference is that `np.arange()` returns an array object:

```
>>> nums = np.arange(1, 10)
>>> nums
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

`np.arange()` starts with the first argument and ends just before the second argument. So, `np.arange(1, 10)` returns an array containing the numbers 1 through 9.

Together, `np.arange()` and `np.reshape()` provide a useful way to create a matrix:

```
>>> matrix = nums.reshape(3, 3)
>>> matrix
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

You can even do this in a single line by chaining together the calls to `np.arange()` and `np.reshape()`:

```
>>> np.arange(1, 10).reshape(3, 3)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

This technique for creating matrices is particularly useful for creating higher-dimensional arrays. Here's how to create a three-dimensional array using `np.array()` and `np.reshape()`:

```
>>> np.arange(1, 13).reshape(3, 2, 2)
array([[[ 1,  2],
        [ 3,  4]],

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

Of course, not every multidimensional array can be built from a sequential list of numbers. In that case, it is often easier to create a flat, one-dimensional list of entries and then `np.reshape()` the array into the desired shape:

```
>>> arr = np.array([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23])
>>> arr.reshape(3, 2, 2)
array([[[ 1,  3],
        [ 5,  7]],

       [[ 9, 11],
        [13, 15]],

       [[17, 19],
        [21, 23]]])
```

In the list passed to `np.array()` in the above example, the difference between any pair of consecutive numbers is 2. You can simplify the creation of these kinds of arrays by passing an optional third argument to `np.arange()` called the **stride**:

```
>>> np.arange(1, 24, 2)
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23])
```

Here's how to rewrite the previous example using `np.arange()` with a stride:

```
>>> np.arange(1, 24, 2).reshape(3, 2, 2)
array([[[ 1,  3],
       [ 5,  7],
       [[ 9, 11],
        [13, 15]],
       [[17, 19],
        [21, 23]]])
```

In this section, you've learned many ways to create and manipulate multidimensional arrays using NumPy's array data structure. But you've only scratched the surface of what you can do with NumPy! At the end of this chapter are some links to further your understanding of NumPy.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Use `np.arange()` and `np.reshape()` to create a 3×3 NumPy array named `A` that includes the numbers 3 through 11.
2. Display the minimum, maximum, and mean of all entries in `A`.
3. Square every entry in `A` using the `**` operator and save the results in an array named `B`.

4. Use `np.vstack()` to stack `a` on top of `b`, then save the results in an array named `c`.
5. Use the `@` operator to calculate the matrix product of `c` by `a`.
6. Reshape `c` into an array of dimensions $3 \times 3 \times 2$.

17.2 Use Matplotlib for Plotting Graphs

In the previous section, you learned how to work with arrays of data using NumPy. While NumPy is great for working with data, people don't typically enjoy looking at large arrays of data. To display data, you need to visualize it into charts and graphs.

In this section, you'll get an introduction to the [Matplotlib](#) package, which is one of the more popular packages for quickly creating two-dimensional figures.

Note

If you've ever created graphs in MATLAB, then you'll find that Matplotlib directly emulates this experience in many ways.

The similarities between MATLAB and Matplotlib are intentional. The MATLAB plotting interface was a direct inspiration for Matplotlib. Even if you haven't used MATLAB, you'll likely find creating plots with Matplotlib to be simple and straightforward.

Let's dive in!

Install Matplotlib

You can install Matplotlib from your terminal with `pip`:

```
$ python3 -m pip install matplotlib
```

You can then view some details about the package with `pip show`:

```
$ python3 -m pip show matplotlib
Name: matplotlib
Version: 3.2.1
Summary: Python plotting package
Home-page: http://matplotlib.org
Author: John D. Hunter, Michael Droettboom
Author-email: matplotlib-users@python.org
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires: python-dateutil, pytz, kiwisolver, numpy,
          cycler, six, pyparsing
Required-by:
```

In particular, note that the latest version at the time of writing was version 3.2.1.

Basic Plotting With pyplot

The Matplotlib package provides two distinct means of creating plots. The first and simplest method is through the `pyplot` interface. This is the interface that MATLAB users will find the most familiar.

The second method for plotting in Matplotlib is through what is known as the [object-oriented API](#). The object-oriented approach offers more control over your plots than is available through the `pyplot` interface. However, the concepts are generally more abstract.

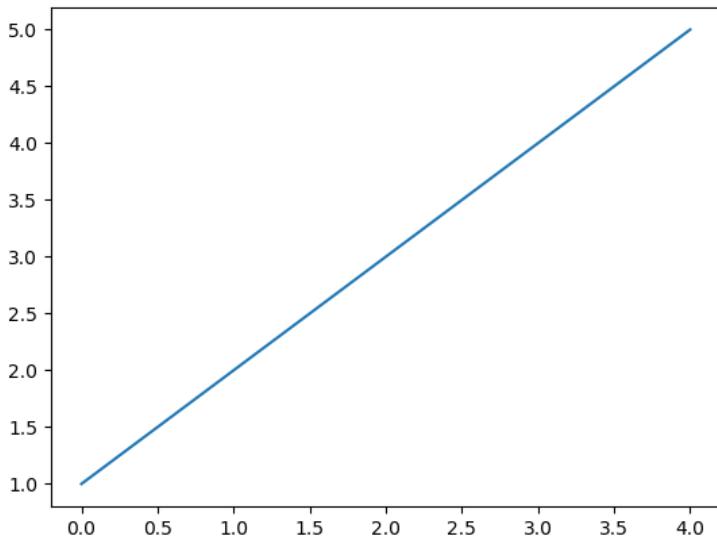
In this section, you'll learn how to get up and running with the `pyplot` interface. You'll be pumping out some great-looking plots in no time!

Let's start by creating a simple plot. Type the following into a new IDLE editor window:

```
from matplotlib import pyplot as plt

plt.plot([1, 2, 3, 4, 5])
plt.show()
```

Save the program and press **F5** to run it. A new window appears displaying the following plot:



`plt.plot([1, 2, 3, 4, 5])` creates a plot with a line through the points $(0, 1)$, $(1, 2)$, $(2, 3)$, $(3, 4)$, and $(4, 5)$.

The list `[1, 2, 3, 4, 5]` passed to `plt.plot()` represents the y-values of the points in the plot. Since you didn't specify any x-values, Matplotlib automatically uses the indices of the list elements which, since Python starts counting at zero, are `0, 1, 2, 3`, and `4`.

The `plt.plot()` function creates a plot, but it doesn't display anything. To display the plot, you need to call `plt.show()`.

You can specify the x-values for the points in your plot by passing two lists to `plt.plot()`. When two arguments are provided to `plt.plot()`, the first list specifies the x-values and the second list specifies the y-values.

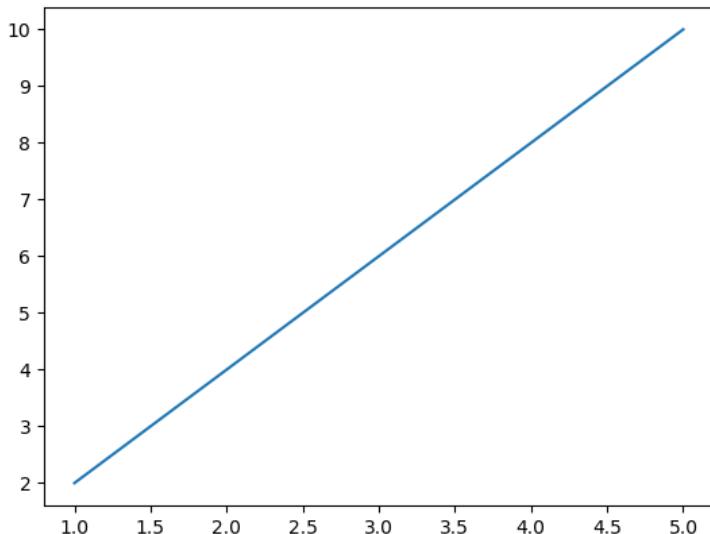
In the editor window, change your program to the following:

```
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
ys = [2, 4, 6, 8, 10]

plt.plot(xs, ys)
plt.show()
```

Save the updated program and press **F5**. You should see the following plot:



Notice that the labels on the axes now reflect the new x- and y-values of the points.

You can use `plot()` to plot more than lines. In the graphs above, the points being plotted just happen to all fall on the same line. By default,

when plotting points with `plot()`, each pair of consecutive points being plotted is connected with a line segment.

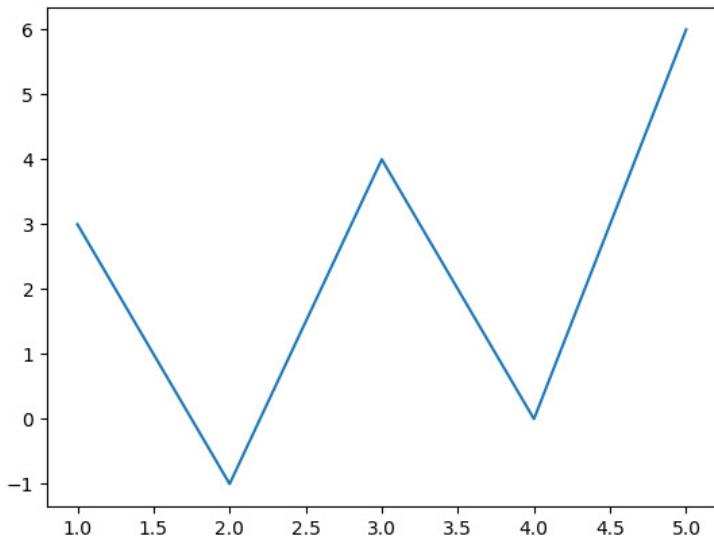
Update your program with the following x- and y-values, which do not fall on a single line:

```
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
ys = [3, -1, 4, 0, 6]

plt.plot(xs, ys)
plt.show()
```

When you save the file and press **F5**, the following plot is displayed:



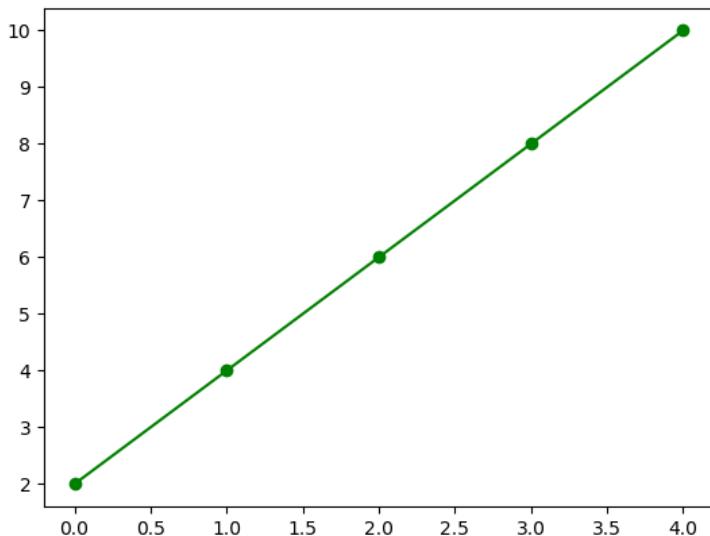
`plot()` has an optional formatting parameter that you can use to specify the color and style of lines or points to draw.

For example, the following program passes the string "g-o" to the formatting parameter:

```
from matplotlib import pyplot as plt

plt.plot([2, 4, 6, 8, 10], "g-o")
plt.show()
```

The g in "g-o" specifies the color green, the - specifies a solid line, and the o specifies circular dots for the points on the line:



You can find the full list of possible formatting combinations in the [Matplotlib documentation](#).

Plot Multiple Graphs in the Same Window

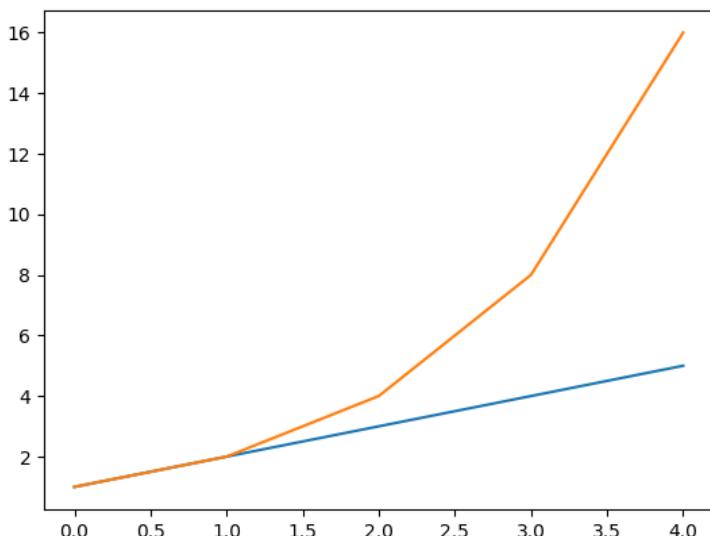
If you need to plot multiple graphs in the same window, then you can call `plot()` multiple times.

For example, the following program displays two plots on the same figure:

```
from matplotlib import pyplot as plt

plt.plot([1, 2, 3, 4, 5])
plt.plot([1, 2, 4, 8, 16])
plt.show()
```

Save the program in a new editor window and press **F5** to see the plot:

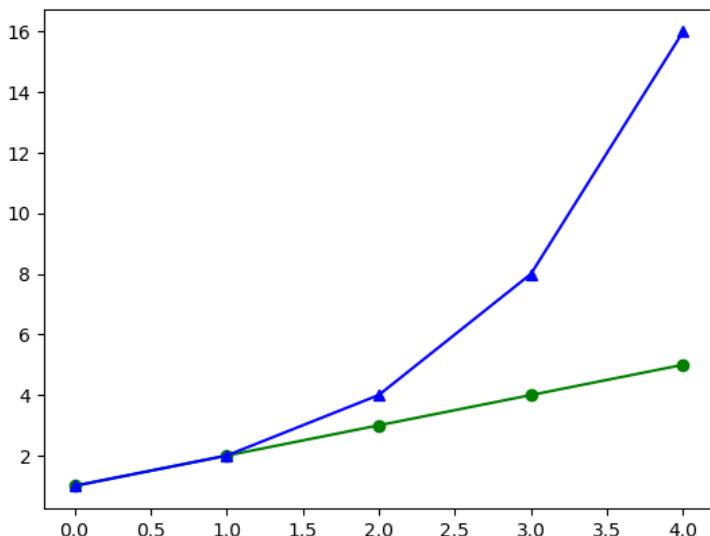


Notice that each graph is displayed in a different color. If you want to control the style of each graph, then you can pass the formatting strings to `plot()` in addition to the x- and y-values:

```
from matplotlib import pyplot as plt

plt.plot([1, 2, 3, 4, 5], "g-o")
plt.plot([1, 2, 4, 8, 16], "b-^")
plt.show()
```

Now the plots get displayed in green and blue with styled points:



Now let's look at some ways to create plots from different types of data sources.

Plot Data From NumPy Arrays

Up to this point, you have been storing your data points in pure Python lists. In the real world, you will most likely be using something like a NumPy array to store your data. Fortunately, Matplotlib plays nicely with array objects.

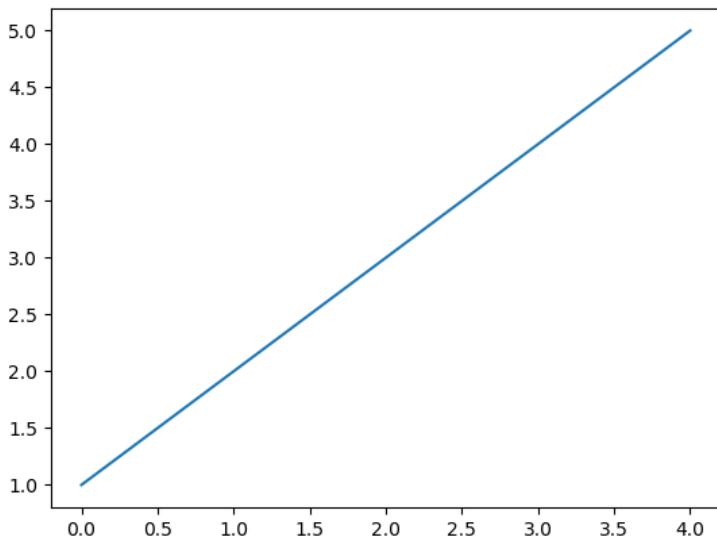
For example, instead of a list, you can use NumPy's `arange()` function to define your data points and then pass the resulting array object to `plot()`:

```
from matplotlib import pyplot as plt
import numpy as np

array = np.arange(1, 6)

plt.plot(array)
plt.show()
```

This produces the following plot:



Passing a two-dimensional array plots each *column* of the array as the y-values for a graph. For example, the following code plots four lines:

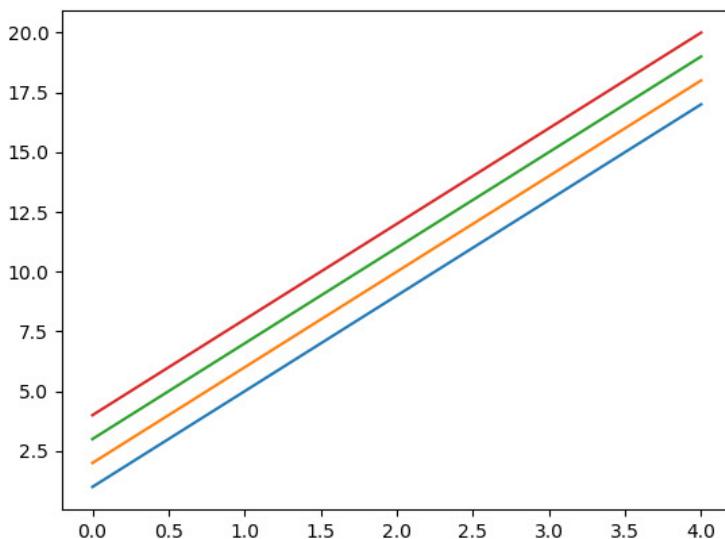
```
from matplotlib import pyplot as plt
import numpy as np

data = np.arange(1, 21).reshape(5, 4)

# data now contains the following array:
# array([[ 1,  2,  3,  4],
#        [ 5,  6,  7,  8],
#        [ 9, 10, 11, 12],
#        [13, 14, 15, 16],
#        [17, 18, 19, 20]])

plt.plot(data)
plt.show()
```

Here are the four lines produced by the above code:



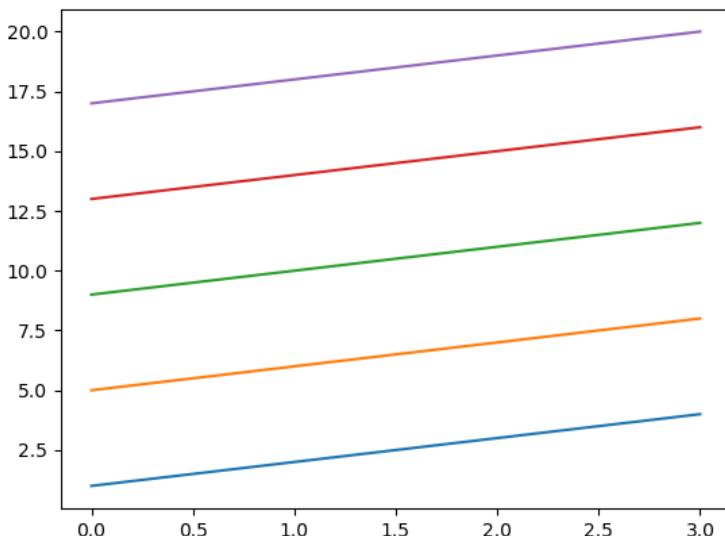
If instead you want to plot the *rows* of the matrix, then you need to plot the **transpose** of the array:

```
from matplotlib import pyplot as plt
import numpy as np

data = np.arange(1, 21).reshape(5, 4)

plt.plot(data.transpose())
plt.show()
```

The plot produced by this program shows the lines created from the rows of the matrix instead of the columns:



So far, the plots you've created don't provide any information about what the plot represents. Next, you'll learn how to format your plots and add text to make them easier to understand.

Format Your Plots to Perfection

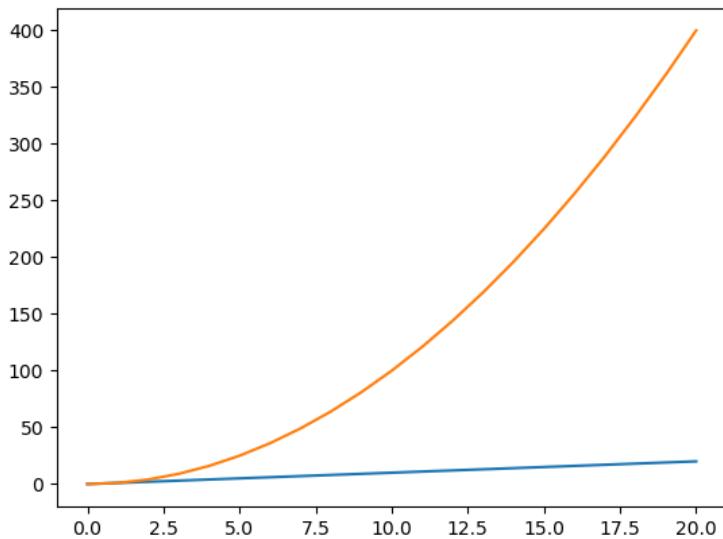
Let's start by plotting the amount of Python learned in the first twenty days of reading Real Python versus another website:

```
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site, real_python = days, days ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.show()
```

The plot produced by this code looks like this:



This plot is less than ideal. The x-axis shows half days instead of full days. There's no title and no text identifying either axis.

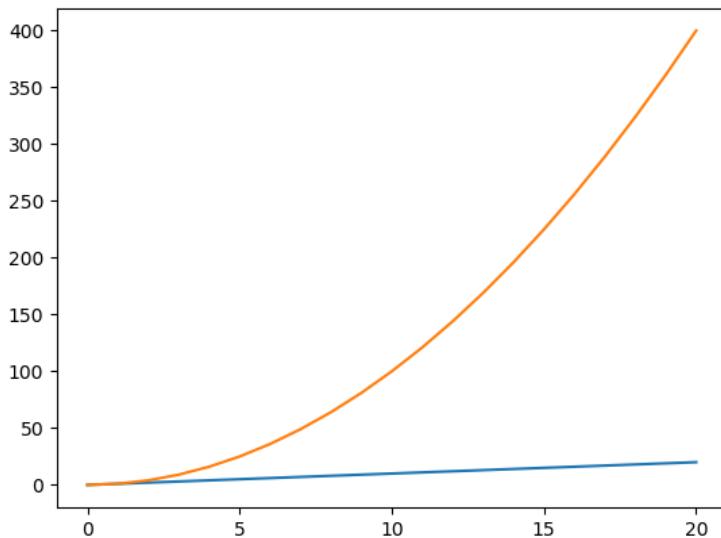
Let's start with adjusting the x-axis. You can use `plt.xticks()` to specify where the ticks should be located:

```
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site, real_python = days, days ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.show()
```

The plot now has ticks on the x-axis marking days 0, 5, 10, 15, and 20:



That's easier to read, but it still isn't clear what each axis represents.

You can add axis labels with the `plt.xlabel()` and `plt.ylabel()` functions. Both functions have a single parameter that expects a string argument containing the axis label.

You can add a title to a plot using `plt.title()`. Like `plt.xlabel()` and `plt.ylabel()`, the `plt.title()` function takes a single string argument containing the title of the plot.

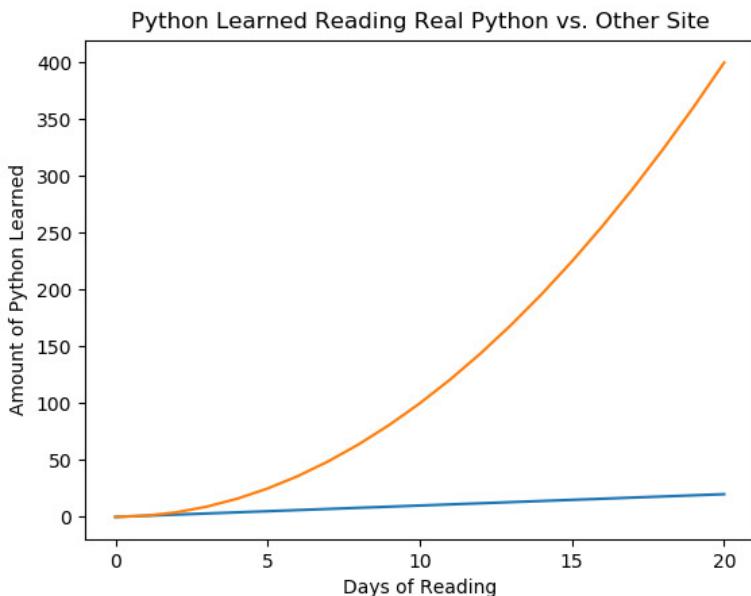
Update the plot code to add the label "Days of Reading" to the x-axis, "Amount of Python Learned" to the y axis, and the title "Python Learned Reading Real Python vs Other Site":

```
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site, real_python = days, days ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.show()
```

Here's the plot with the title and labeled axes:



Now we're starting to get somewhere!

But there's still one problem: it's not clear which graph represents Real Python and which represents the other website. To clarify which graph is which, you can add a legend with `plt.legend()`.

`plt.legend()` has one required positional parameter. You pass to it a list of strings containing the name for each plot in the figure. The strings need to be in the same order that you add each plot to the figure.

For example, the following updated version of the plot code adds a legend that identifies the plot for "Other Site" and the plot for "Real Python".

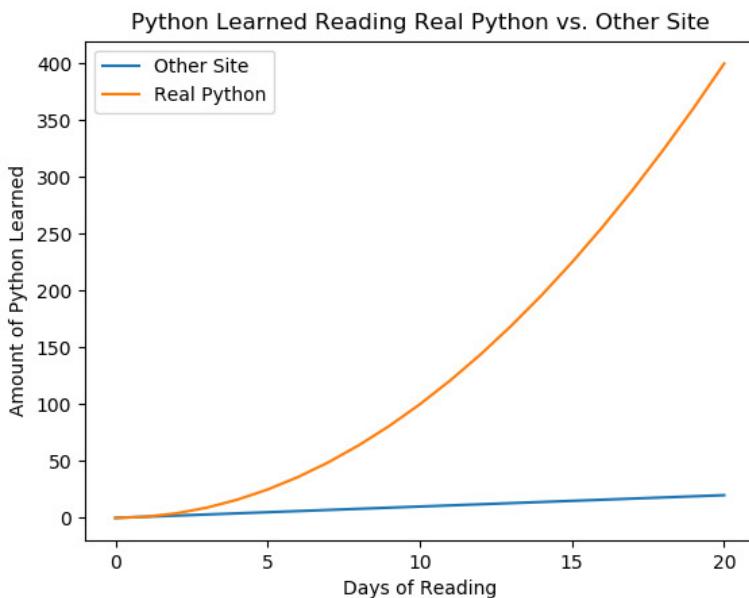
Since you add the plot with the data for `other_site` to the figure first, the first string in the list you pass to `plt.legend()` is "Other Site":

```
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site, real_python = days, days ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.legend(["Other Site", "Real Python"])
plt.show()
```

Here's what the final figure looks like:



`plt.legend()` has a number of optional parameters that you can use to customize the legend. Check out the [legend guide](#) in the Matplotlib documentation for more information.

Other Types of Plots

So far you've only created line charts, but Matplotlib can produce many different kinds of charts, including bar charts and histograms.

Bar Charts

You can create bar charts using the `plt.bar()` function. `plt.bar()` has two required parameters:

1. A list of x-values for the center point for each bar
2. A list of y-values for the height of each bar

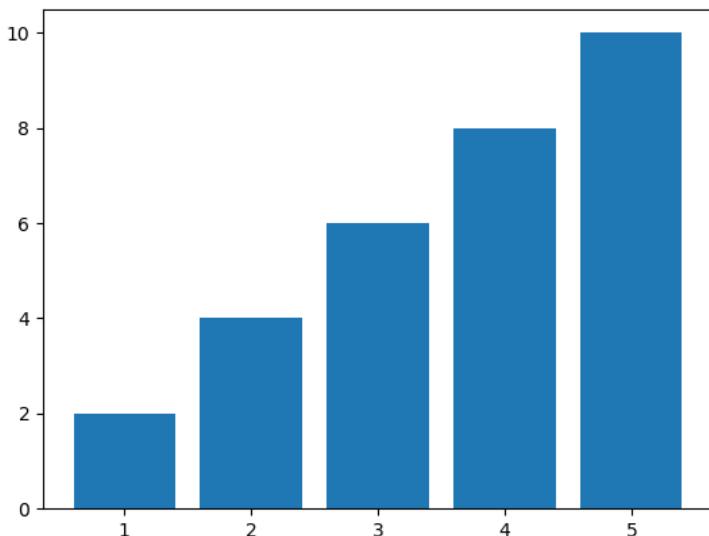
For example, the following code produces a bar chart with bars centered at points 1, 2, 3, 4, and 5 on the x-axis with heights of 2, 4, 6, 8, and 10, respectively:

```
from matplotlib import pyplot as plt

centers = [1, 2, 3, 4, 5]
tops = [2, 4, 6, 8, 10]

plt.bar(centers, tops)
plt.show()
```

Here's what the bar chart looks like:



You can use a NumPy array instead of a list to specify the center points and bar heights. The following code produces a plot identical to the previous one using NumPy arrays instead of lists:

```
from matplotlib import pyplot as plt
import numpy as np

centers = np.arange(1, 6)
tops = np.arange(2, 12, 2)

plt.bar(centers, tops)
plt.show()
```

`plt.bar()` is quite flexible. The first argument doesn't need to be a list of numbers. It could be a list of strings representing categories of data.

Suppose you wanted to plot a bar chart representing the data contained in the following dictionary:

```
fruits = {  
    "apples": 10,  
    "oranges": 16,  
    "bananas": 9,  
    "pears": 4,  
}
```

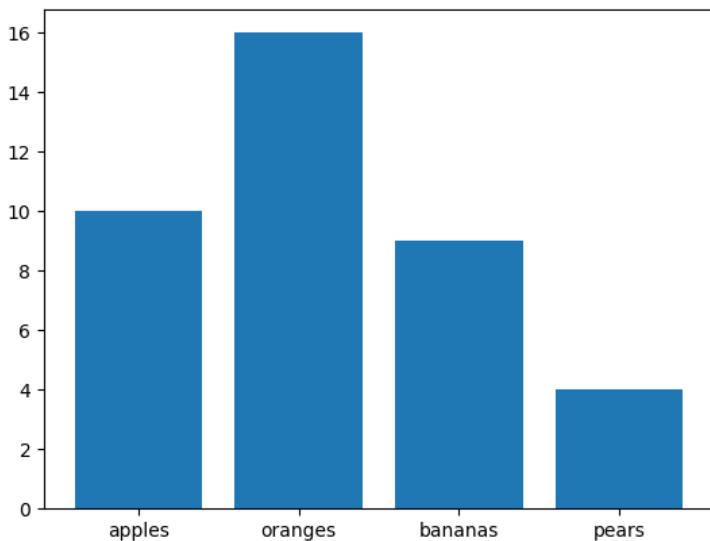
You can get a list of the names of the fruits using `fruits.keys()` and the corresponding values using `fruits.values()`:

```
>>> fruits.keys()  
dict_keys(['apples', 'oranges', 'bananas', 'pears'])  
  
>>> fruits.values()  
dict_values([10, 16, 9, 4])
```

You can pass `fruits.keys()` and `fruits.values()` to `plt.bar()` to plot the values against the fruit names in a bar chart:

```
from matplotlib import pyplot as plt  
  
fruits = {  
    "apples": 10,  
    "oranges": 16,  
    "bananas": 9,  
    "pears": 4,  
}  
  
plt.bar(fruits.keys(), fruits.values())  
plt.show()
```

The bars are evenly spaced, and the names of the fruits are conveniently used as the tick labels along the x-axis:



Histograms

Another commonly used type of graph is the [histogram](#), which shows how data is distributed. Histograms are created in Matplotlib using the `plt.hist()` function.

`plt.hist()` has two required parameters:

1. A list (or NumPy array) of values
2. The number of bins to display in the histogram

`plt.hist()` can calculate the frequency of each value in the list of values and calculate the bins for you. This saves you a ton of trouble trying to make a histogram with a standard bar chart.

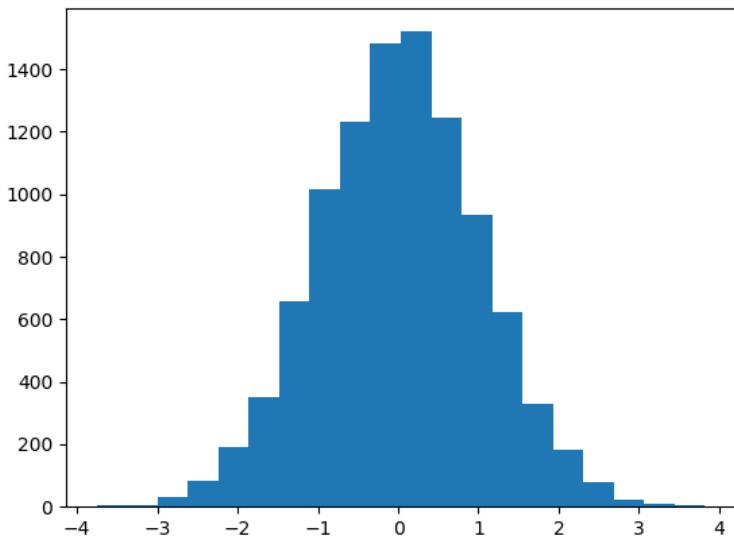
Let's look at an example that plots the histogram for ten thousand normally distributed random numbers grouped into twenty bins. To create the random numbers, we'll use the `randn()` function from NumPy's `random` module. This returns an array of random floating-point numbers, most of which are close to zero.

Here's the code for the histogram:

```
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)
plt.show()
```

Matplotlib automatically creates twenty evenly spaced bins with a width of 0.5:



Histograms are highly customizable. For a detailed discussion of creating histograms with Python, check out [Real Python's “Python Histogram Plotting: NumPy, Matplotlib, Pandas & Seaborn.”](#)

Save Figures as Images

You may have noticed that the window displaying your plots has a toolbar at the bottom. You can use this toolbar to save your plot as an image file.

More often than not, you probably won't want to sit at your computer and click the Save button for each plot that you want to export. Fortunately, Matplotlib makes it easy to save your plots programmatically.

To save your plot, use the `plt.savefig()` function. Pass the path to where you would like to save your plot as a string. The example below saves a simple bar chart as `bar.png` to the current working directory. If you'd like to save to somewhere else, then you need to provide an absolute path:

```
from matplotlib import pyplot as plt
import numpy as np

xs = np.arange(1, 6)
tops = np.arange(2, 12, 2)

plt.bar(xs, tops)
plt.savefig("bar.png")
```

Note

If you want to both save a figure and display it on the screen, make sure that you save it first before you display it!

The `show()` function pauses execution of your code, and closing the display window destroys the graph, so trying to save the figure after calling `show()` results in an empty file.

Work With Plots Interactively

When you're initially tweaking the layout and formatting of a particular graph, it can be helpful to change parts of the graph without having to rerun an entire program just to see the results.

One of the easiest ways to do this is with a [Jupyter Notebook](#), which creates an interactive Python interpreter session that runs in your browser.

Jupyter Notebooks have become a staple for interacting with and exploring data, and they work great with both NumPy and Matplotlib.

For an interactive tutorial on how to use Jupyter Notebooks, check out Jupyter's "[IPython in Depth](#)" tutorial.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Re-create as many of the graphs shown in this section as you can by writing your own programs without referring to the provided code.
2. Do pirates cause global warming? In the chapter 17 `practice_files` folder is a CSV file with data about the number of pirates and the global temperature. Write a program that visually examines this relationship by reading the `pirates.csv` file and graphing the number of pirates along the x-axis and the temperature along the y-axis. Add a title and label the graph's axes, then save the resulting graph as a PNG image file.

17.3 Summary and Additional Resources

In this chapter, you learned about scientific computing and data visualization in Python. You saw how to:

- Work with arrays and matrices using NumPy
- Create plots with Matplotlib

It would take an entire book to do justice to the topics of scientific computing, data analysis, and data visualization. The topics that you learned in this chapter provide a solid foundation for starting your journey into scientific computing, data analysis, and data science.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-scientific-computing

Additional Resources

To learn more, check out the following resources:

- “Look Ma, No For-Loops: Array Programming With NumPy”
- Data Science With Python Core Skills (Learning Path)

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 18

Graphical User Interfaces

Throughout this book, you've been creating **command-line applications**, which are programs that are started from and produce output in a terminal window.

Command-line apps are fine for making tools that you or other developers might use, but the vast majority of software users never want to open a terminal!

A **graphical user interface**, or **GUI** (pronounced “gooey”) for short, has windows with components like buttons and text fields that give users a familiar, visual way to interact with a program.

In this chapter, you'll learn how to:

- Add a simple GUI to a command-line application with EasyGUI
- Create full-featured GUI applications with Tkinter

Let's get started!

18.1 Add GUI Elements With EasyGUI

You can use the EasyGUI library to quickly add a graphical user interface to your program. EasyGUI is somewhat limited, but it works well for simple tools that need just a little bit of input from the user.

In this section, you'll use EasyGUI to create a short GUI program that allows a user to pick a PDF file from their hard drive and rotate its pages by a selected amount.

Install EasyGUI

To get started, you need to install EasyGUI with pip:

```
$ python3 -m pip install easygui
```

Once EasyGUI is installed, you can check out some details of the package with pip show:

```
$ python3 -m pip show easygui
Name: easygui
Version: 0.98.1
Summary: EasyGUI is a module for very simple, very easy GUI
        programming in Python. EasyGUI is different from other
        GUI generators in that EasyGUI is NOT event-driven.
        Instead, all GUI interactions are invoked by simple
        function calls.

Home-page: https://github.com/robertlugg/easygui
Author: easygui developers and Stephen Ferg
Author-email: robert.lugg@gmail.com
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

The code in this chapter is written using EasyGUI version 0.98.1, the same version you see in the information shown above.

Your First EasyGUI Application

EasyGUI is great for displaying **dialog boxes** to collect user input and display output. It's not particularly great for creating a large application with several windows, menus, and toolbars.

You can think of EasyGUI as a sort of replacement for the `input()` and `print()` functions that you've been using for input and output.

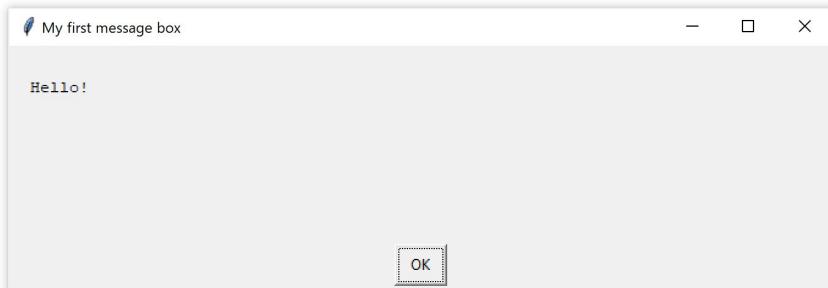
Program flow with EasyGUI typically works like this:

1. At some point in the code, a visual element is displayed on the user's screen.
2. Execution of the code pauses until the user provides input through the visual element.
3. The user's input is returned as an object and execution of the code resumes.

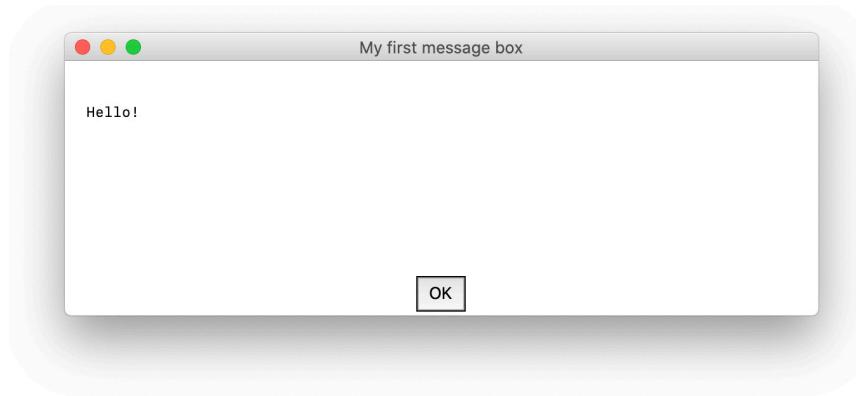
To get a feel for how EasyGUI works, open a new interactive window in IDLE and execute the following lines of code:

```
>>> import easygui as gui  
>>> gui.msgbox(msg="Hello!", title="My first message box")
```

If you run the code on Windows, then you'll see a window like the following displayed on your screen:



The window's appearance depends on the operating system on which the code is executed. On macOS, the window looks like this:



Here's what the window looks like on Ubuntu:



The examples in this section will use Windows screenshots.

Important

Both EasyGUI and IDLE are written using the Tkinter library, which you'll learn about in the next section. This overlap sometimes causes issues with execution, such as dialog boxes getting frozen or stuck.

If you think this might be happening to you, try running your code from a terminal. You can start an interactive Python session from a terminal with the `python` command on Windows and `python3` on macOS/Ubuntu.

Let's break down what you see in the dialog box that you generated with the code above:

1. The string "`Hello!`", passed to the `msg` parameter of `msgbox()`, is displayed as the message in the message box.
2. The string "`My first message box`" passed to the `title` parameter is displayed as the title of the message box.
3. The message box contains a button labeled "OK".

Press `OK` to close the dialog box and look at IDLE's interactive window. The string "OK" is displayed below the last line of code that you typed:

```
>>> gui.msgbox(msg="Hello!", title="My first message box")
'OK'
```

`msgbox()` returns the button label when the dialog box is closed. If the user closes the dialog box without pressing `OK`, then `msgbox()` returns the value `None`.

You can customize the button label by setting a third optional parameter called `ok_button`. For example, the following creates a message box with a button labeled "Click me":

```
>>> gui.msgbox(msg="Hello!", title="Greeting", ok_button="Click me")
```

`msgbox()` is great for displaying a message, but it doesn't provide the user with many options for interacting with your program. EasyGUI has several functions that display various types of dialog boxes. Let's explore some of these now!

EasyGUI's Ensemble of GUI Elements

Besides `msgbox()`, EasyGUI has several other functions for displaying different kinds of dialog boxes. The following table summarizes some of the available functions:

Function	Description
<code>msgbox()</code>	Displays a message with a single button and returns the label of the button
<code>buttonbox()</code>	Displays a message with several buttons and returns the label of the selected button
<code>indexbox()</code>	Displays a message with several buttons and returns the index of the selected button
<code>enterbox()</code>	Prompts the user with a text entry box and returns the text entered by the user
<code>fileopenbox()</code>	Prompts the user to select a file to be opened and returns the absolute path to the selected file
<code>diropenbox()</code>	Prompts the user to select a directory to be opened and returns the absolute path to the selected directory
<code>filesavebox()</code>	Prompts the user for a location to save a file and returns the absolute path to the save location

Let's look at each one of these individually.

`buttonbox()`

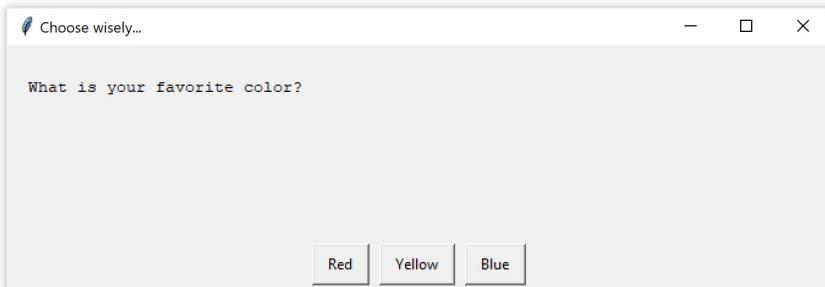
EasyGUI's `buttonbox()` displays a dialog box with a message and several buttons that the user can click. The label of the clicked button is returned to your program.

Just like `msgbox()`, `buttonbox()` has `msg` and `title` parameters for setting the message to be displayed and the title of the dialog box. `buttonbox()` also has a third parameter called `choices` that is used to set up the buttons.

For example, the following code produces a dialog box with three buttons labeled "Red", "Yellow", and "Blue":

```
>>> gui.buttonbox(  
...     msg="What is your favorite color?",  
...     title="Choose wisely...",  
...     choices=("Red", "Yellow", "Blue"),  
... )
```

Here's what the dialog box looks like:



When you press one of the buttons, the button label is returned as a string. For example, pressing `Yellow` causes `buttonbox()` to return the string "Yellow":

```
>>> gui.buttonbox(  
...     msg="What is your favorite color?",  
...     title="Choose wisely...",  
...     choices=("Red", "Yellow", "Blue"),  
... )  
'Yellow'
```

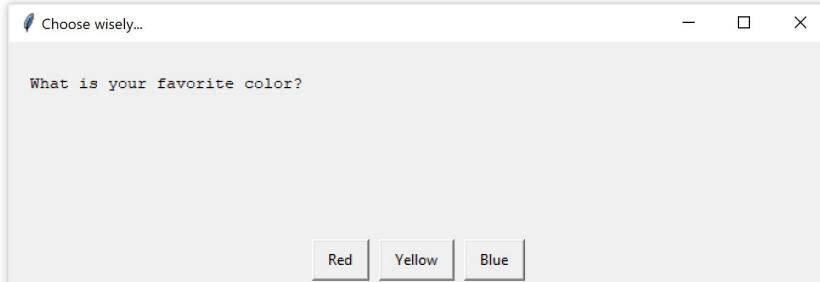
Just like `msgbox()`, `buttonbox()` returns the value `None` if the user closes the dialog box without pressing one of the buttons.

`indexbox()`

`indexbox()` displays a dialog box that looks identical to the dialog box displayed by `buttonbox()`. In fact, you create an `indexbox()` the same way you create a `buttonbox()`:

```
>>> gui.indexbox(  
...     msg="What's your favorite color?",  
...     title="Choose wisely...",  
...     choices=("Red", "Yellow", "Blue"),  
... )
```

Here's what the dialog box looks like:



The difference between `indexbox()` and `buttonbox()` is that `indexbox()` returns the index of the button label in the list or tuple passed to `choices` instead of the label itself.

For example, if you click `Yellow`, then the integer `1` is returned:

```
>>> gui.indexbox(  
...     msg="What's your favorite color?",  
...     title="Favorite color",  
...     choices=("Red", "Yellow", "Blue"),  
... )  
1
```

Because `indexbox()` returns an index and not a string, it's a good idea to define the tuple for `choices` outside the function so you can reference the label by index later in your code:

```
>>> colors = ("Red", "Yellow", "Blue")  
>>> choice = gui.indexbox(  
...     msg="What's your favorite color?",  
...     title="Favorite color",  
...     choices=colors,  
)  
>>> choice  
1  
>>> colors[choice]  
'Yellow'
```

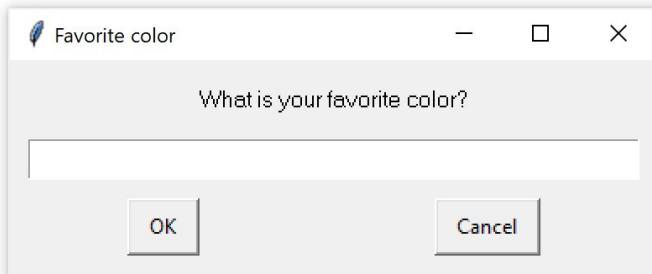
`buttonbox()` and `indexbox()` are great for getting input from a user when they need to choose from a predetermined set of choices. These functions are not well suited for getting information such as a user's name or email address. For that, you can use the `enterbox()`.

enterbox()

`enterbox()` is used to collect text input from a user:

```
>>> gui.enterbox(  
...     msg="What is your favorite color?",  
...     title="Favorite color",  
... )
```

The dialog box produced by `enterbox()` has an input box in which the user can type their own answer:



Type in a color name, such as `Yellow`, and press `OK`. The text you entered is returned as a string:

```
>>> gui.enterbox(  
...     msg="What is your favorite color?",  
...     title="Favorite color",  
... )  
'Yellow'
```

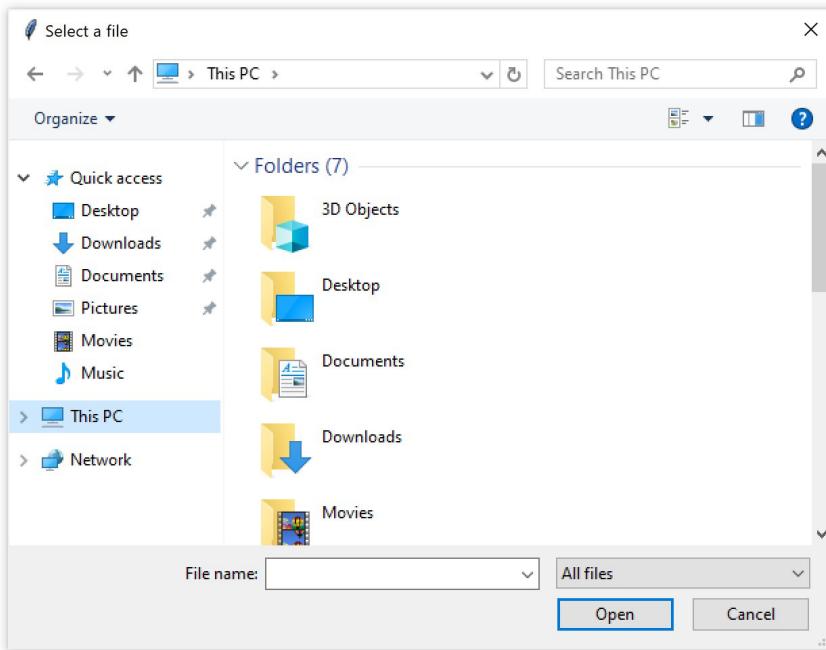
One of the most common reasons for displaying a dialog box is to allow a user to select a file or folder in their file system. EasyGUI has some special functions designed just for these operations.

`fileopenbox()`

`fileopenbox()` displays a dialog box for selecting a file to be opened:

```
>>> gui.fileopenbox(title="Select a file")
```

The dialog box looks like the standard system file open dialog box:



Select a file and click **Open**. A string containing the full path to the selected file is returned.

Important

`fileopenbox()` doesn't actually open the file! To do that you need to use the built-in `open()` like you learned in chapter 12.

Just like `msgbox()` and `buttonbox()`, `fileopenbox()` returns the value `None` if the user presses `Cancel` or closes the dialog box without selecting a file.

diropenbox() and filesavebox()

EasyGUI has two other functions that generate dialogs nearly identical to the one generated by `fileopenbox()`:

1. `diropenbox()` opens a dialog that can be used to select a folder instead of a file. When the user presses `Open`, the full path to the directory is returned.
2. `filesavebox()` opens a dialog to select a location for saving a file and will confirm that the user wants to overwrite the file if the chosen name already exists. Just like `fileopenbox()`, `filesavebox()` returns the file path when the user presses `Save`. The file is not actually saved.

Important

Neither `diropenbox()` nor `filesavebox()` actually opens a directory or saves a file. They only return the absolute path to the directory to be opened or the file to be saved.

To open the directory or save the file, you need to write the code yourself.

Both `diropenbox()` and `filesavebox()` return `None` if the user closes the dialog without pressing `Open` or `Save`. This can cause your program to crash if you aren't careful.

For example, the following raises a `TypeError` if the user closes the dialog box without making a selection:

```
>>> path = gui.fileopenbox(title="Select a file")
>>> open_file = open(path, "r")
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
TypeError: expected str, bytes or os.PathLike object, not NoneType
```

How you handle situations like this has a huge impact on a user's experience with your program.

Exiting Your Program Gracefully

Suppose you're writing a program for extracting pages from a PDF file. The first thing the program might do is use `fileopenbox()` so that the user can select a PDF to open.

What do you do if the user decides they don't want to run the program and presses `Cancel`?

You must make sure that your program handles these situations gracefully. The program shouldn't crash or produce any unexpected output. In the situation described above, the program should probably just stop running altogether.

One way to stop a program from running is to use Python's built-in `exit()` function.

For example, the following program uses `exit()` to stop the program when the user presses `Cancel` in a file selection dialog box:

```
import easygui as gui

path = gui.fileopenbox(title="Select a file")

if path is None:
    exit()
```

If the user closes the dialog box without pressing `OK`, then `path` is `None` and the program executes `exit()` in the `if` block. The program closes and execution stops.

Note

If you're running the program in IDLE, then `exit()` also closes the current interactive window. It's very thorough.

The `is` keyword used in the above code example is something you haven't seen before in this book. `is` compares two objects and determines whether or not they are the same object by checking whether or not they have the memory address.

Recall that you can get the memory address of an object with the `id()` function. So, if two objects `id(a)` and `id(b)` are the same, for two objects `a` and `b`, then `a is b` evaluates to `True`. In this case, `a` and `b` aren't two distinct objects. They represent the same object with two different names.

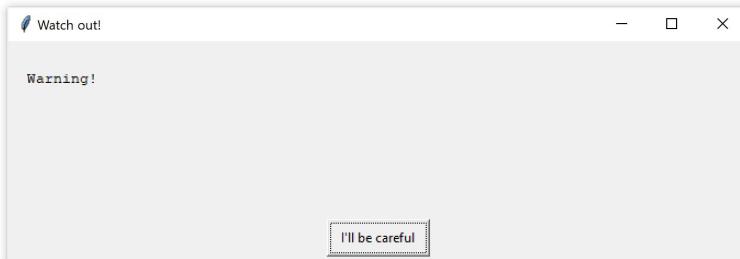
The reason that you use `is` to compare a value to `None` is because there can only be one `None` object. Whenever a function returns `None`, it returns the same object in memory as the one referred to by the `None` keyword.

Now that you know how to create dialog boxes with EasyGUI, let's put everything together into a real-world application.

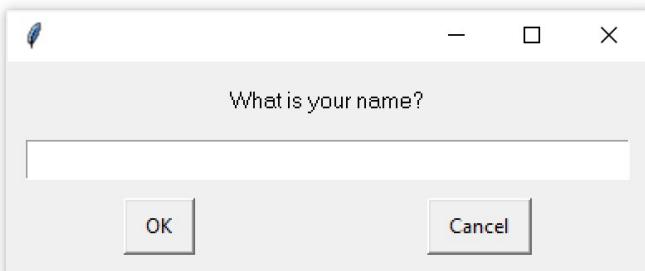
Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Create the following dialog box:



2. Create the following dialog box:



18.2 Example App: PDF Page Rotator

EasyGUI is a great choice for utility applications that automate simple yet repetitive tasks. If you work in an office, then you can really boost your productivity by creating tools with EasyGUI that take the pain out of everyday to-do items.

In this section, you'll use some of the EasyGUI dialog boxes you learned about in the last section to create an application for rotating PDF pages.

The Application Design

Before you dive into the code, put some thought into how the program should work.

The program needs to ask the user which PDF file to open, by how many degrees they want to rotate each page, and where they would like to save the new PDF. Then the program needs to open the file, rotate the pages, and save the new file.

Note

When you're designing an application, it helps to plan out each step before you start coding. For large applications, drawing diagrams describing the program flow can help keep everything organized.

You can map this out into explicit steps that you can more easily translate into code:

1. Display a file selection dialog for opening a PDF file.
2. If the user cancels the dialog, then exit the program.
3. Ask the user to select one of 90, 180, or 270 degrees.
4. Display a file selection dialog for saving the rotated PDF.
5. If the user tries to save a file with the same name as the input file:
 - Alert the user with a message box that this is not allowed.
 - Return to step 4.
6. If the user cancels the file save dialog, then exit the program.
7. Perform the page rotation:
 - Open the selected PDF.
 - Rotate all the pages.
 - Save the rotated PDF to the selected file.

Implement the Design

Now that you have a plan, you can tackle each step one at a time. Open a new editor window in IDLE.

First, import EasyGUI and PyPDF2:

```
import easygui as gui  
from PyPDF2 import PdfFileReader, PdfFileWriter
```

Step 1 in your plan is to display a file selection dialog for opening a PDF file. You can do this with `fileopenbox()`:

```
# 1. Display a file selection dialog for opening a PDF file.  
input_path = gui.fileopenbox(  
    title="Select a PDF to rotate...",  
    default="*.pdf"  
)
```

Here, you've set the `default` parameter to `"*.pdf"`, which configures the dialog to display only files with the `.pdf` extension. This helps prevent the user from accidentally selecting a file that isn't a PDF.

The file path selected by the user is assigned to the `input_path` variable. If the user closes the dialog without selecting a file path (step 2), then `input_path` is `None`.

To exit the program when the user closes the dialog box without selecting a value, check that `input_path` is `None` and, if so, call `exit()`:

```
# 2. If the user cancels the dialog, then exit the program.  
if input_path is None:  
    exit()
```

The third step is to ask the user how much they'd like to rotate the PDF pages. They can choose either 90, 180, or 270 degrees.

You can use a `buttonbox()` to collect this information:

```
# 3. Ask the user to select one of 90, 180, or 270 degrees.  
choices = ("90", "180", "270")  
degrees = gui.buttonbox(  
    msg="Rotate the PDF clockwise by how many degrees?",  
    title="Choose rotation...",  
    choices=choices,  
)
```

The dialog generated here has three buttons with the labels "90", "180", and "270". When the user clicks on one of these buttons, the label of the button is assigned to the `degrees` variable as a string.

To rotate the pages in the PDF by the selected angle, you'll need the value to be an integer, not a string. Go ahead and convert it to an integer:

```
degrees = int(degrees)
```

Next, get the output file path from the user using `filesavebox()`:

```
# 4. Display a file selection dialog for saving the rotated PDF.  
save_title = "Save the rotated PDF as..."  
file_type = "*.pdf"  
output_path = gui.filesavebox(title=save_title, default=file_type)
```

As with `fileopenbox()`, you've set the `default` parameter to `*.pdf`. This ensures that the file automatically gets saved with the `.pdf` extension.

The user shouldn't be allowed to overwrite the original file (step 5). You can use a `while` loop to repeatedly show the user a warning until they pick a path that's different from the input file path:

```
# 5. If the user tries to save with the same name as the input file:  
while input_path == output_path:  
    # - Alert the user with a message box that this is not allowed.  
    gui.msgbox(msg="Cannot overwrite original file!")  
    # - Return to step 4.  
    output_path = gui.filesavebox(title=save_title, default=file_type)
```

The `while` loop checks if `input_path` is the same as `output_path`. If it isn't, then the loop body is ignored. If `input_path` and `output_path` are the same, then `msgbox()` shows a warning to the user telling them they can't overwrite the original file.

After warning the user, `fileSaveBox()` displays another file save dialog box with the same title and default file type as before. This is the part that returns the user to step 4. Even though the program doesn't actually return the line of code where `fileSaveBox()` is first called, the effect is the same.

If the user closes the file save dialog without pressing `Save`, then the program should exit (step 6):

```
# 6. If the user cancels the file save dialog, then exit the program.  
if output_path is None:  
    exit()
```

Now you have everything you need to implement the last step of the program:

```
# 7. Perform the page rotation:  
#     - Open the selected PDF.  
input_file = PdfFileReader(input_path)  
output_pdf = PdfFileWriter()  
  
#     - Rotate all the pages.  
for page in input_file.pages:  
    page = page.rotateClockwise(degrees)  
    output_pdf.addPage(page)  
  
#     - Save the rotated PDF to the selected file.  
with open(output_path, "wb") as output_file:  
    output_pdf.write(output_file)
```

Try out your new PDF rotation application! It works equally well on Windows, macOS, and Ubuntu Linux!

Here's the full application source code for your reference:

```
import easygui as gui
from PyPDF2 import PdfFileReader, PdfFileWriter

# 1. Display a file selection dialog for opening a PDF file.
input_path = gui.fileopenbox(
    title="Select a PDF to rotate...",
    default="*.pdf"
)

# 2. If the user cancels the dialog, then exit the program.
if input_path is None:
    exit()

# 3. Ask the user to select one of 90, 180 or 270 degrees.
choices = ("90", "180", "270")
degrees = gui.buttonbox(
    msg="Rotate the PDF clockwise by how many degrees?",
    title="Choose rotation...",
    choices=choices,
)

# 4. Display a file selection dialog for saving the rotated PDF.
save_title = "Save the rotated PDF as..."
file_type = "*.pdf"
output_path = gui.filesavebox(title=save_title, default=file_type)

# 5. If the user tries to save with the same name as the input file:
while input_path == output_path:
    # - Alert the user with a message box that this is not allowed.
    gui.msgbox(msg="Cannot overwrite original file!")
    # - Return to step 4.
    output_path = gui.filesavebox(title=save_title, default=file_type)
```

```
# 6. If the user cancels the file save dialog, then exit the program.  
if output_path is None:  
    exit()  
  
# 7. Perform the page rotation:  
#     - Open the selected PDF.  
input_file = PdfFileReader(input_path)  
output_pdf = PdfFileWriter()  
  
#     - Rotate all the pages.  
for page in input_file.pages:  
    page = page.rotateClockwise(degrees)  
    output_pdf.addPage(page)  
  
#     - Save the rotated PDF to the selected file.  
with open(output_path, "wb") as output_file:  
    output_pdf.write(output_file)
```

EasyGUI is great for quickly creating a GUI for small tools and applications. For larger projects, EasyGUI may be too limited. That's where Python's built-in [Tkinter](#) library comes in.

Tkinter is a GUI framework that operates at a lower level than EasyGUI. That means you have more control over the visual aspects of the GUI, such as window size, font size, font color, and what GUI elements are present in a dialog box or window.

The rest of this chapter is devoted to developing GUI applications with Python's built-in Tkinter library.

Review Exercise

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. The GUI application for rotating PDF pages in this section has a problem. The program crashes if the user closes the `buttonbox()` used to select degrees without selecting a value.

Fix this problem by using a `while` loop to keep displaying the selection dialog if `degrees` is `None`.

18.3 Challenge: PDF Page Extraction Application

In this challenge, you'll use EasyGUI to write a GUI application for extracting pages from a PDF file.

Here's a detailed plan for the application:

1. Ask the user to select a PDF file to open.
2. If no PDF file is chosen, then exit the program.
3. Ask for a starting page number.
4. If the user doesn't enter a starting page number, then exit the program.
5. Valid page numbers are positive integers. If the user enters an invalid page number:
 - Warn the user that the entry is invalid.
 - Return to step 3.
6. Ask for an ending page number.
7. If the user doesn't enter an ending page number, then exit the program.
8. If the user enters an invalid page number:
 - Warn the user that the entry is invalid.
 - Return to step 6.
9. Ask for the location to save the extracted pages.
10. If the user doesn't select a save location, then exit the program.

11. If the chosen save location is the same as the input file path:
 - Warn the user that they can't overwrite the input file.
 - Return to step 9.
12. Perform the page extraction:
 - Open the input PDF file.
 - Write a new PDF file containing only the pages in the selected page range.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

18.4 Introduction to Tkinter

Python has [a lot](#) of GUI frameworks, but Tkinter is the only framework that's built into the Python standard library.

Tkinter has several strengths. It's **cross-platform**, meaning the same code works on Windows, macOS, and Linux. Visual elements are rendered using native operating system elements, so applications built with Tkinter look like they belong no matter which platform you run them on.

Although Tkinter is considered the de facto Python GUI framework, it's not without criticism. One notable criticism is that GUIs built with Tkinter look outdated. If you want a shiny, modern interface, then Tkinter may not be what you're looking for.

However, Tkinter is lightweight and relatively simple to use compared with other frameworks. This makes it a compelling choice for building GUI applications in Python, especially when a modern sheen is unnecessary and quickly building something that is functional and cross-platform is the top priority.

Note

As mentioned in the last section, IDLE is built with Tkinter. You may encounter difficulties when running your own GUI programs within IDLE.

If you find that the GUI window you're trying to create is unexpectedly freezing or making IDLE behave in some unexpected way, then try running your program from the command prompt or terminal.

Let's dive right in and see how you build an application with Tkinter.

Your First Tkinter Application

The foundational element of a Tkinter GUI is the **window**. Windows are the containers in which all other GUI elements live. Other GUI elements, such as text boxes, labels, and buttons, are known as **widgets**. Widgets are contained inside windows.

Let's create a window that contains a single widget. Start by opening a new interactive window in IDLE.

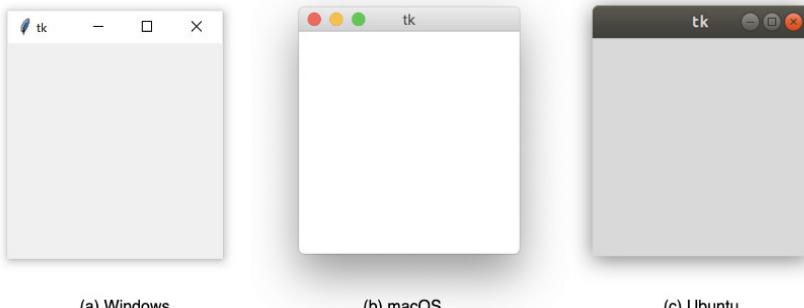
The first thing you need to do is import the Tkinter module:

```
>>> import tkinter as tk
```

A window is an instance of Tkinter's `Tk` class. Go ahead and create a new window and assign it to the variable `window`:

```
>>> window = tk.Tk()
```

When you execute the above code, a new window pops up on your screen. How it looks depends on your operating system:



For the rest of this chapter, Windows screenshots will be used.

Now that we have a window, let's add a widget. The `tk.Label` class is used to add some text to a window.

Create a `Label` widget with the text "Hello, Tkinter" and assign it to a variable called `greeting`:

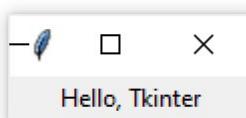
```
>>> greeting = tk.Label(text="Hello, Tkinter")
```

The window you created earlier doesn't change. You just created a `Label` widget, but it hasn't been added to the window yet.

There are several ways to add widgets to a window. Right now, we'll use the `Label` widget's `.pack()` method:

```
>>> greeting.pack()
```

The window now looks like this:



When you `.pack()` a widget into a window, Tkinter sizes the window as small as it can while still fully encompassing the widget.

Now execute the following:

```
>>> window.mainloop()
```

Nothing seems to happen, but notice that no new prompt appears in the shell.

`window.mainloop()` tells Python to run the Tkinter application and **blocks** any code that comes after it from running until the window it's called on is closed. Go ahead and close the window you've created, and you'll see a new prompt displayed in the shell.

Important

When you work with Tkinter from a REPL like IDLE's interactive window, updates to windows are applied as each line is executed.

This isn't the case when a Tkinter program is executed from a Python file.

If you don't include `window.mainloop()` at the end of a program in a Python file, then the Tkinter application will never run, and nothing will be displayed.

Creating a window with Tkinter takes only a couple of lines of code. But blank windows aren't very useful! In the next section, you'll learn about some of the widgets available in Tkinter and how you can customize them to meet your application's needs.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Using Tkinter from IDLE's interactive window, execute code that creates a window with a `Label` widget displaying the text "GUIs are great!"

2. Repeat exercise 1 with the text "Python rocks!"
3. Repeat exercise 1 with the text "Engage!"

18.5 Working With Widgets

Widgets are the bread and butter of Tkinter. They're the elements through which users interact with your program.

Each widget in Tkinter is defined by a class. Here are some of the widgets available:

Widget Class	Description
Label	A widget used to display text on the screen
Button	A button that can contain text and can perform an action when clicked
Entry	A text entry widget that allows only a single line of text
Text	A text entry widget that allows multiline text entry
Frame	A rectangular region used to group related widgets or provide padding between widgets

You'll see how to work with each of these in the following sections.

Note

Tkinter has many more widgets than the ones listed here. For a full list, check out the “[Basic Widgets](#)” and “[More Widgets](#)” articles in the [TkDocs](#) tutorial.

Let's take a closer look at the `Label` widget.

Label Widgets

Label widgets are used to display text or images. The text displayed by a Label widget can't be edited by the user. It's for display purposes only.

As you saw in the example at the beginning of this chapter, you can create a Label widget by instantiating the Label class and passing a string to the text parameter:

```
label = tk.Label(text="Hello, Tkinter")
```

Label widgets display text with the default system text color and background color. These are typically black and white, respectively, but you may see different colors if you've changed those settings in your operating system.

You can control Label text and background colors using the foreground and background parameters:

```
label = tk.Label(  
    text="Hello, Tkinter",  
    foreground="white", # Set the text color to white  
    background="black" # Set the background color to black  
)
```

There are numerous valid color names, including:

- "red"
- "orange"
- "yellow"
- "green"
- "blue"
- "purple"

Many of the [HTML color names](#) work with Tkinter.

Note

You can find a full list of colors, including macOS- and Windows-specific system colors that are controlled by the current system theme, on the [TkDocs site](#).

You can also specify a color using hexadecimal RGB values:

```
label = tk.Label(text="Hello, Tkinter", background="#34A2FE")
```

This sets the label background to a nice light blue color.

Hexadecimal RGB values are more cryptic than named colors, but they are more flexible. Fortunately, there are [tools](#) available that make getting hexadecimal color codes relatively painless.

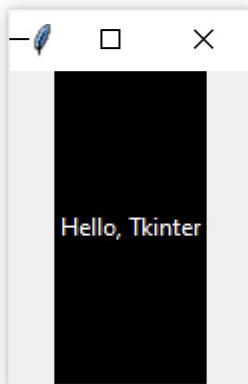
If you don't feel like typing out `foreground` and `background` all the time, then you can use the shorthand `fg` and `bg` parameters to set the foreground and background colors:

```
label = tk.Label(text="Hello, Tkinter", fg="white", bg="black")
```

You can also control the width and height of a label with the `width` and `height` parameters:

```
label = tk.Label(  
    text="Hello, Tkinter",  
    fg="white",  
    bg="black",  
    width=10,  
    height=10  
)
```

Here's what this label looks like in a window:



It may seem strange that the label in the window isn't square even though `width` and `height` are both set to 10. This is because the height and width are measured in **text units**.

One horizontal text unit is determined by the width of the character o (the number zero) in the default system font. Similarly, one vertical text unit is determined by the height of the character o.

Note

To ensure consistent behavior of the application across platforms, Tkinter uses text units for width and height measurements instead of inches, centimeters, or pixels.

Measuring units by the width of a character means that the size of a widget will be relative to the default font on a user's machine. This ensures that text will fit properly in labels and buttons no matter where the application is running.

Labels are great for displaying text, but they don't help you get input from a user. The next three widgets that you'll look at are all used to get user input.

Button Widgets

Button widgets are used to display clickable buttons. They can be configured to call a function whenever they're clicked. We'll talk about how to call functions from button clicks in the next section. For now, let's look at how to create and style a Button.

There are many similarities between `Button` and `Label` widgets. In many ways, a `Button` is just a `Label` that you can click! The same keyword arguments used to create and style a `Label` work with `Button` widgets.

For example, the following code creates a `Button` with a blue background, yellow text, and height and width set to 25 and 5 text units, respectively:

```
button = tk.Button(  
    text="Click me!",  
    width=25,  
    height=5,  
    bg="blue",  
    fg="yellow",  
)
```

Here's what the `Button` looks like in a window:



Pretty nifty!

Note

Button backgrounds **do not work** on macOS. It is a limitation of the operating system and not a bug in Tkinter.

The next two widgets you'll see are used to collect text input from a user.

Entry Widgets

When you need to get a little bit of text from a user, like a name or an email address, use an `Entry` widget. It displays a small text box that the user can type into.

You create and style an `Entry` widget pretty much the same as `Label` and `Button` widgets. For example, the following creates a widget with a blue background, yellow text, and a width of 50 text units:

```
entry = tk.Entry(fg="yellow", bg="blue", width=50)
```

The interesting bit about `Entry` widgets isn't how you style them, though. It's how you use them get input from a user. There are three main operations that you can perform with `Entry` widgets:

1. Retrieving text with `.get()`
2. Deleting text with `.delete()`
3. Inserting text with `.insert()`

The best way to get a grip on `Entry` widgets is to create one and interact with it. Go ahead and open IDLE's interactive window and follow along with the examples in this section.

First, import `tkinter` and create a new window:

```
>>> import tkinter as tk  
>>> window = tk.Tk()
```

Now create a `Label` and an `Entry` widget:

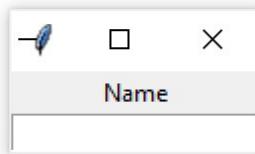
```
>>> label = tk.Label(text="Name")
>>> entry = tk.Entry()
```

The `Label` describes what sort of text should go in the `Entry` widget. It doesn't enforce any sort of requirements on the `Entry`, but it tells the user what your program expects them to put there.

Next, you need to `.pack()` the widgets into the window so that they're visible:

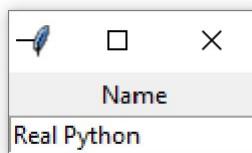
```
>>> label.pack()
>>> entry.pack()
```

Here's what that looks like:



Notice that Tkinter automatically centers the `Label` above the `Entry` widget in the window. This is a feature of the `.pack()` method, which you'll learn more about in later sections.

Click inside the `Entry` widget and type "Real Python":



Now you've got some text entered into the `Entry` widget, but that text hasn't been sent to your program yet.

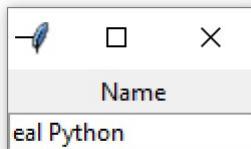
Use the `Entry` widget's `.get()` method to retrieve the text and assign it to a variable called `name`:

```
>>> name = entry.get()
>>> name
'Real Python'
```

You can delete text using the `Entry` widget's `.delete()` method. Passing an integer argument to `.delete()` tells it which character to remove. For example, `.delete(0)` deletes the first character from the `Entry`:

```
>>> entry.delete(0)
```

The text remaining in the widget is now "eal Python":



Note

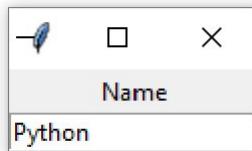
Just like Python string objects, text in an `Entry` widget is indexed starting from 0.

If you need to remove several characters from an `Entry`, pass a second integer argument to `.delete()` indicating the index of the character at which deletion should stop.

For example, the following deletes the first four letters in the revised `Entry`:

```
>>> entry.delete(0, 4)
```

This command deletes the characters "e", "a", and "l" along with the space. The remaining text now reads "Python":



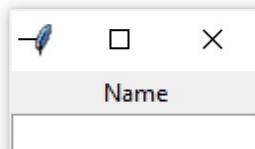
Note

`Entry.delete()` works just like string slices. The first argument determines the starting index and the deletion continues up to *but not including* the index passed as the second argument.

Use the special constant `tk.END` as the second argument of `.delete()` to remove all text in an `Entry`:

```
>>> entry.delete(0, tk.END)
```

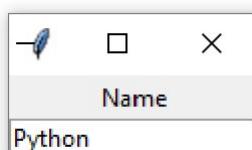
You now have a blank text box:



To insert text into an `Entry` widget, use the `.insert()` method:

```
>>> entry.insert(0, "Python")
```

The window now looks like this:



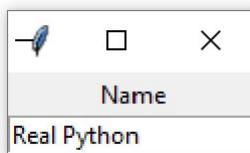
The first argument tells `.insert()` where to insert the text. If there's no text in the `Entry`, then the new text will always be inserted at the beginning of the widget, no matter what value you pass to the first argument.

For example, if you had called `.insert()` above with `100` as the first argument instead of `0`, then it would have generated the same output.

If an `Entry` already contains text, then `.insert()` will insert the new text at the specified position and shift all existing text to the right:

```
>>> entry.insert(0, "Real ")
```

The widget text now reads "Real Python":



`Entry` widgets are great for capturing small amounts of text from a user, but because they're always displayed on a single line, they're not ideal for gathering large amounts of text. That's where `Text` widgets come in!

Text Widgets

`Text` widgets are used for entering text, just like `Entry` widgets. The difference is that `Text` widgets can contain multiple lines of text.

With a `Text` widget, a user can input a whole paragraph—or even several pages—of text!

Just like `Entry` widgets, there are three main operations you can perform with `Text` widgets:

1. Retrieve text with `.get()`
2. Delete text with `.delete()`
3. Insert text with `.insert()`

Although the method names are the same as the `Entry` methods, they work a bit differently. Let's get our hands dirty by creating a `Text` widget and seeing what all it can do.

Note

If you still have the window open from the previous section, then you can close it by executing the following in IDLE's interactive window:

```
>>> window.destroy()
```

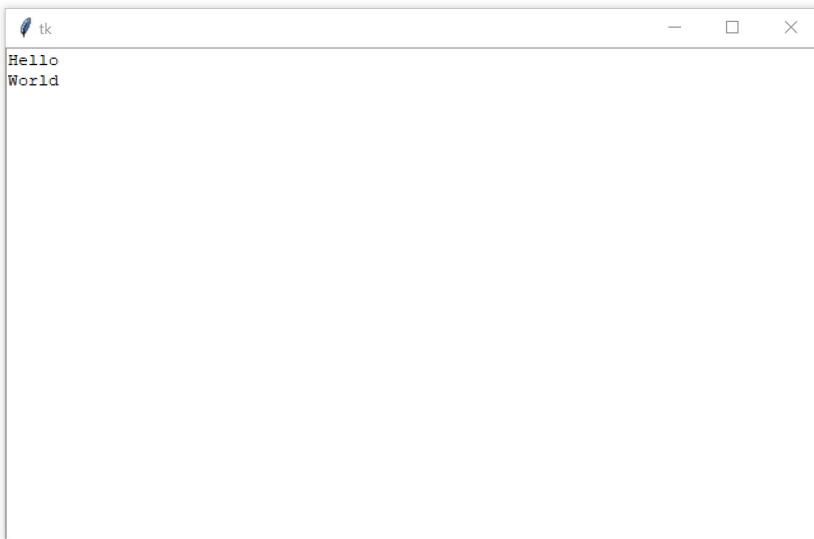
You can also close the window manually by clicking  on the window itself.

In IDLE's interactive window, create a new blank window and `.pack()` a `Text` widget into it:

```
>>> window = tk.Tk()  
>>> text_box = tk.Text()  
>>> text_box.pack()
```

A window with a text box should appear on your screen. Click anywhere inside the window to activate the text box. Type in the word "Hello", then press `Enter` and type "World" on the second line.

The window should look like this:



Just like `Entry` widgets, you can retrieve the text from a `Text` widget using `.get()`. However, calling `.get()` without arguments doesn't return all the text in the text box like it does for `Entry` widgets. It raises an exception:

```
>>> text_box.get()
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    text_box.get()
TypeError: get() missing 1 required positional argument: 'index1'
```

`Text.get()` requires at least one argument. Calling `.get()` with a single index returns a single character. To retrieve several characters, you need to pass two arguments: a start index and an end index.

Indices in `Text` widgets work differently from `Entry` widgets. Since `Text` widgets can have several lines of text, an index must contain two pieces of information:

1. The line number of a character
2. The position of a character on that line

Line numbers start counting from 1 and character positions start counting from 0.

To make an index, you create a string of the form "<line>.<char>", replacing <line> with the line number and <char> with the character number.

For example, "1.0" represents the first character on the first line. "2.3" represents the fourth character on the second line.

Let's use the index "1.0" to get the first letter from the first line of the text box that you created earlier:

```
>>> text_box.get("1.0")
'H'
```

Since character indices start from 0 and the word "Hello" starts at the first position in the text box, the index of the letter o is 4. Just like Python string slices, to get the entire word Hello from the text box, the end index must be one higher than the index of the last character to be read.

So, to get the entire word "Hello" from the text box, you use "1.0" for the first index and "1.5" for the second index:

```
>>> text_box.get("1.0", "1.5")
'Hello'
```

To get the word "World" on the second line of the text box, change the line number in each index to 2:

```
>>> text_box.get("2.0", "2.5")
'World'
```

To get all the text in a text box, set the starting index to "1.0" and use the special tk.END constant for the second index:

```
>>> text_box.get("1.0", tk.END)
'Hello\nWorld\n'
```

Notice that text returned by `.get()` includes newline characters. You can also see from this example that every line in a `Text` widget has a newline character at the end, including the last line.

The `.delete()` method is used to delete characters from a text box. It works just like the `.delete()` method for `Entry` widgets.

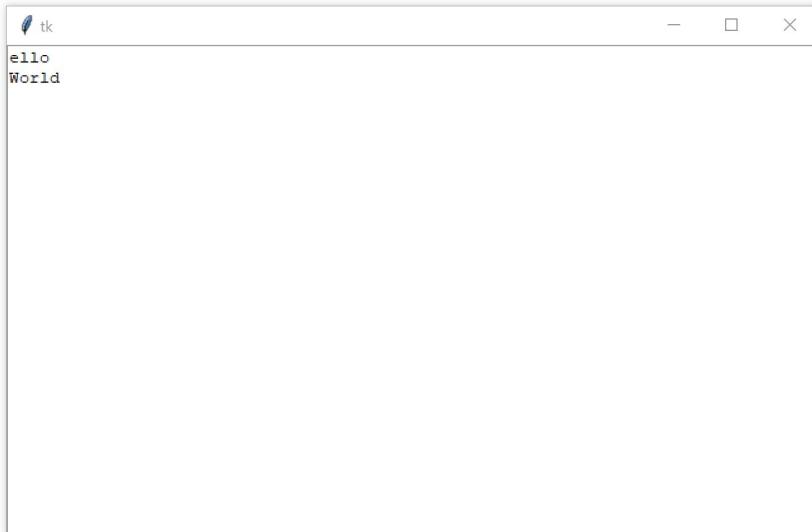
There are two ways to use the `.delete()` method:

1. With a single argument
2. With two arguments

In the single argument version, you pass to `.delete()` the index of a single character to be deleted. For example, the following deletes the first character, `H`, from the text box:

```
>>> text_box.delete("1.0")
```

The first line of text in the window now reads "ello":

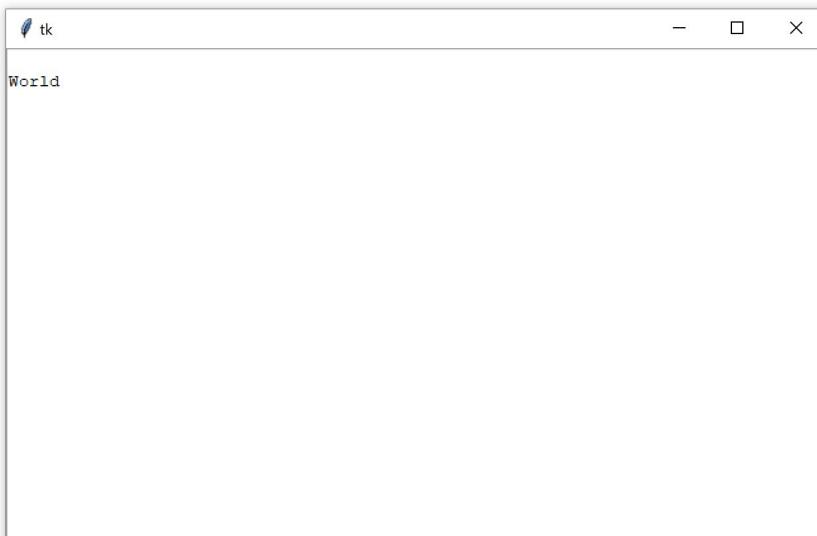


In the two-argument version, you pass two indices to delete a range of characters starting at the first index and going up to but not including the second index.

For example, to delete the remaining "ello" on the first line of the text box, use the indices "1.0" and "1.4":

```
>>> text_box.delete("1.0", "1.4")
```

Notice that the text is gone from the first line, leaving a blank line followed the word "World" on the second line:



Even though you can't see it, there's still a character on the first line—the newline character!

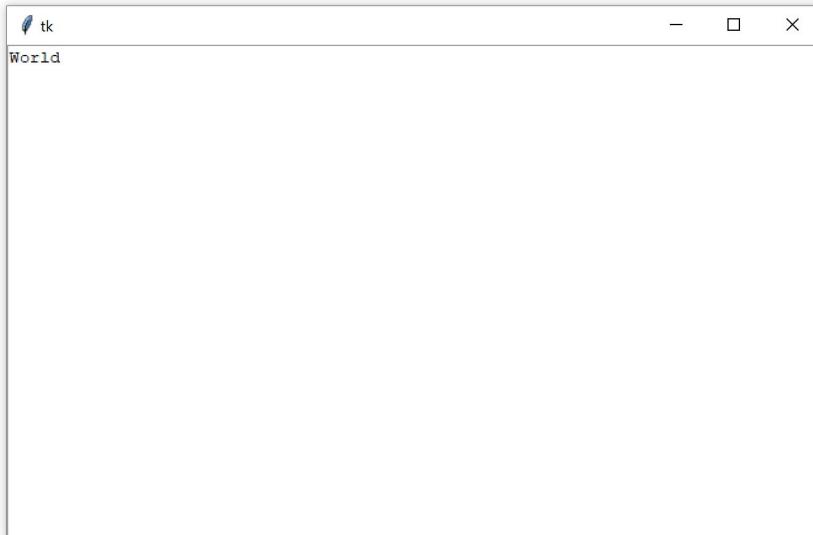
You can verify this using `.get()`:

```
>>> text_box.get("1.0")
'\n'
```

If you delete that character, then the remaining contents of the text box will shift up a line:

```
>>> text_box.delete("1.0")
```

Now "World" is on the first line of the text box:



Let's clear out the rest of the text in the text box. Set "1.0" as the start index and use `tk.END` for the second index:

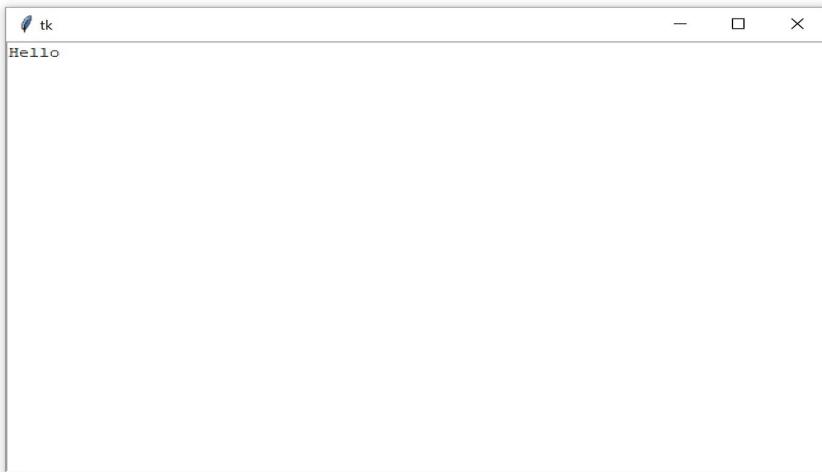
```
>>> text_box.delete("1.0", tk.END)
```

The text box is now empty.

You can insert text into a text box using `.insert()`:

```
>>> text_box.insert("1.0", "Hello")
```

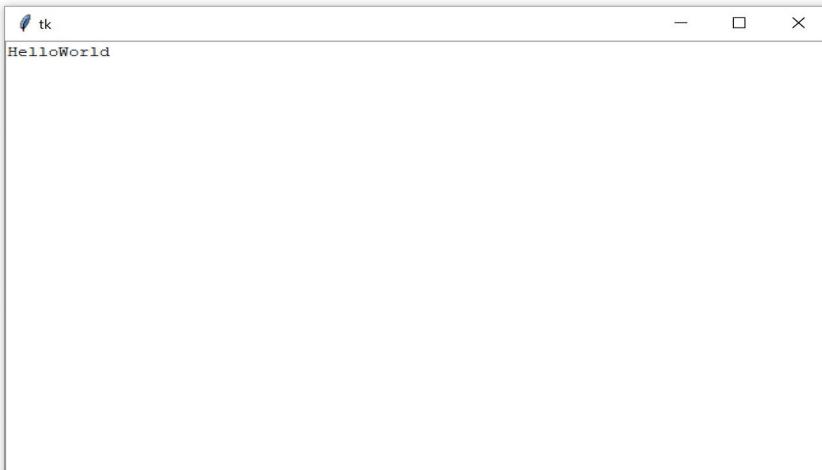
This inserts the word "Hello" at the beginning of the text box:



Check out what happens if you try to insert the word "World" on the second line:

```
>>> text_box.insert("2.0", "World")
```

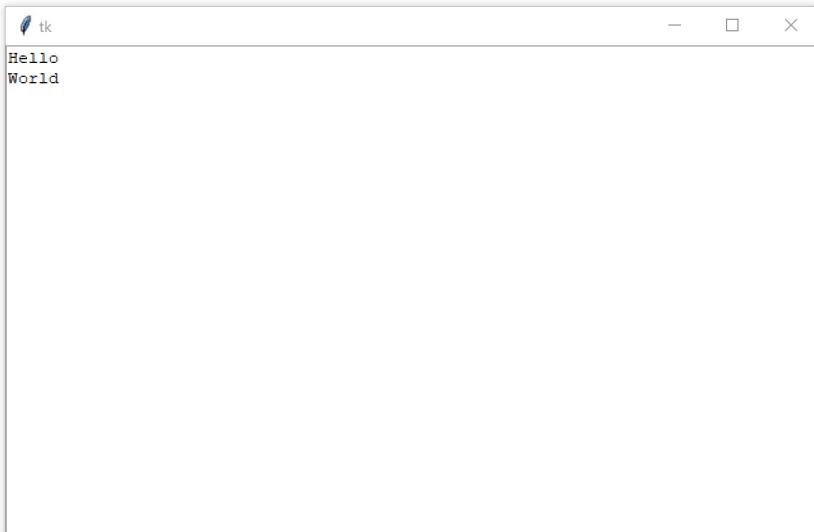
Instead of being inserted on the second line, the text is inserted at the end of the first line:



If you want to insert text into a new line, then you need to manually insert a newline character into the string being inserted:

```
>>> text_box.insert("2.0", "\nWorld")
```

Now "World" is on the second line of the text box:



So, `.insert()` will either insert text at the specified position if there's already text at that position, or it will append text to the specified line if the character number is greater than the index of the last character in the text box.

It's usually impractical to try to keep track of the index of the last character. The best way to insert text at the end of a `Text` widget is to pass `tk.END` to the first parameter of `.insert()`:

```
text_box.insert(tk.END, "Put me at the end!")
```

Don't forget to include the newline character (`\n`) at the beginning of the text if you want to put it on a new line:

```
text_box.insert(tk.END, "\nPut me on a new line!")
```

`Label`, `Button`, `Entry`, and `Text` widgets are just a few of the widgets available in Tkinter. There are several others, including widgets for checkboxes, radio buttons, scroll bars, and progress bars. For more information on all the other available widgets, check out the [tutorial on TkDocs.com](#).

In this chapter, we're going to work with only five widgets: the four you've seen so far plus the `Frame` widget. `Frame` widgets are important for organizing the layout of the widgets in your application.

Before we get into the details of laying out the visual presentation of your widgets, let's take a closer look at how `Frame` widgets work and how you can assign other widgets to them.

Assigning Widgets to Frames

The following program creates a blank `Frame` widget and assigns it to the main application window:

```
import tkinter as tk

window = tk.Tk()
frame = tk.Frame()
frame.pack()

window.mainloop()
```

`frame.pack()` packs the frame into the window so that the window sizes itself as small as possible while still encompassing the frame.

When you run the above code, you get some seriously uninteresting output:



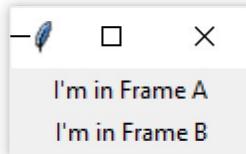
An empty `Frame` widget is practically invisible. Frames are best thought of as containers for other widgets. You can assign a widget to a frame by setting the widget's `master` attribute:

```
frame = tk.Frame()  
label = tk.Label(master=frame)
```

To get a feel for how this works, let's write a program that creates two `Frame` widgets called `frame_a` and `frame_b`. You want `frame_a` to contain a label with the text "I'm in Frame A" and `frame_b` to contain the label "I'm in Frame B". Here's one way to do that:

```
import tkinter as tk  
  
window = tk.Tk()  
  
frame_a = tk.Frame()  
frame_b = tk.Frame()  
  
label_a = tk.Label(master=frame_a, text="I'm in Frame A")  
label_a.pack()  
  
label_b = tk.Label(master=frame_b, text="I'm in Frame B")  
label_b.pack()  
  
frame_a.pack()  
frame_b.pack()  
  
window.mainloop()
```

Notice that `frame_a` is packed into the window before `frame_b`. The window that opens shows the label in `frame_a` above the label in `frame_b`:



Now let's see what happens when you swap the order of `frame_a.pack()` and `frame_b.pack()`:

```
import tkinter as tk

window = tk.Tk()

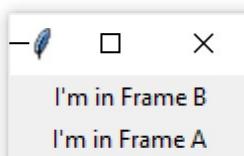
frame_a = tk.Frame()
label_a = tk.Label(master=frame_a, text="I'm in Frame A")
label_a.pack()

frame_b = tk.Frame()
label_b = tk.Label(master=frame_b, text="I'm in Frame B")
label_b.pack()

# Order of `frame_a` and `frame_b` is swapped
frame_b.pack()
frame_a.pack()

window.mainloop()
```

The output looks like this:



Now `label_b` is on top. Since `label_b` was assigned to `frame_b`, it moves to wherever you position `frame_b`.

All four of the widget types you've learned about—`Label`, `Button`, `Entry`, and `Text`—have a `master` attribute that is set when you instantiate them. That way you can control which `Frame` a widget is assigned to.

`Frame` widgets are great for organizing other widgets in a logical manner. You can assign related widgets to the same frame so that if the frame is ever moved in the window, then the related widgets stay together.

In addition to grouping your widgets logically, `Frame` widgets can add a little flair to the visual presentation of your application. Read on to see how to create various borders for `Frame` widgets.

Adjusting Frame Appearance With Reliefs

`Frame` widgets can be configured with a `relief` attribute that creates a border around the frame. You can set `relief` to any of the following values:

- `tk.FLAT` creates no border effect. This is the default value.
- `tk.SUNKEN` creates a sunken effect.
- `tk.RAISED` creates a raised effect.
- `tk.GROOVE` creates a grooved border effect.
- `tk.RIDGE` creates a ridged effect.

To apply the border effect, you must set the `borderwidth` attribute to a value greater than 1. This attribute adjusts the width of the border in pixels.

The best way to get a feel for what each relief effect does is to see them in action for yourself. Here's a program that packs five `Frame` widgets into a window, each with a different value for the `relief` argument:

```
import tkinter as tk

border_effects = {
    "flat": tk.FLAT,
    "sunken": tk.SUNKEN,
    "raised": tk.RAISED,
    "groove": tk.GROOVE,
    "ridge": tk.RIDGE,
}

window = tk.Tk()

for relief_name, relief in border_effects.items():
    # 1
    frame = tk.Frame(master=window, relief=relief, borderwidth=5)
    # 2
    frame.pack(side=tk.LEFT)
    # 3
    label = tk.Label(master=frame, text=relief_name)
    label.pack()

window.mainloop()
```

Let's break that code down.

First, you create a dictionary and assign it to the `border_effects` variable. The keys of the dictionary are the names of the different relief effects available in Tkinter, and the values are the corresponding Tkinter objects.

Next, after creating the `window` object, you use a `for` loop to loop over each item in the `border_effects` dictionary. At each step in the loop, you perform three operations:

1. You create a new `Frame` widget and assign it to the `window` object. You set the `relief` attribute to the corresponding relief effect in the `border_effects` dictionary, and you set the `borderwidth` attribute to 5 so

that the effect is visible.

2. You pack the `Frame` into the window using `.pack()`. The `side` keyword argument tells Tkinter which direction to pack the `frame` objects. You'll see more on how this works in the next section.
3. You create a `Label` widget to display the name of the relief and pack it into the `frame` object you just created.

The window produced by the above code looks like this:



The image shows examples of each relief style:

- `tk.FLAT` creates a flat-looking frame.
- `tk.SUNKEN` adds a border that gives the frame the appearance of being sunken into the window.
- `tk.RAISED` gives the frame a border that makes it appear to protrude from the screen.
- `tk.GROOVE` adds a border that appears as a sunken groove around an otherwise flat frame.
- `tk.RIDGE` gives the appearance of a raised lip around the edge of the frame.

Widget Naming Conventions

When you create a widget, you can give it any name you like as long as it's a valid Python identifier. It's usually a good idea, though, to include the name of the widget class in the variable name that you assign to the widget instance.

For example, if you use a `Label` widget to display a user's name, then you might name the widget `label_user_name`. An `Entry` widget used to collect a user's age might be called `entry_user_age`.

When you include the widget class name in the variable name, you help yourself and anyone else that needs to read your code understand what type of widget the variable name refers to.

Using the full name of the widget class can lead to long variable names, so you may want to adopt a shorthand for referring to each widget type. For the rest of this chapter, we'll use the following shorthand prefixes to name widgets:

Widget Class	Prefix	Example Name
Label	lbl	lbl_name
Button	btn	btn_submit
Entry	ent	ent_age
Text	txt	txt_notes
Frame	frm	frm_address

In this section, you learned how to create a window, use widgets, and work with frames. At this point, you can make some simple windows that display messages, but a full-blown application is still out of reach.

In the next section, you'll learn how to control the layout of your applications using Tkinter's powerful geometry managers.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Try to re-create all the screenshots in this section without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for ten or fifteen minutes and try again.

Repeat this until you can produce all the screenshots on your own. Focus on the output. It's okay if your own code is slightly different from the code in the book.

2. Write a program that displays a `Button` widget that is fifty text units wide and twenty-five text units tall. It should have a white background with blue text that reads "Click here".
3. Write a program that displays an `Entry` widget that is forty text units wide and has a white background and black text. Use `.insert()` to display text in the `Entry` widget that reads "What is your name?"

18.6 Controlling Layout With Geometry Managers

Until now, you've been adding widgets to windows and `Frame` widgets using `.pack()`, but you haven't learned exactly what this method does. Let's clear things up!

In Tkinter, you control the layout of your application with **geometry managers**. The `.pack()` method is an example of a geometry manager, but it isn't the only one. Tkinter has two others: `.place()` and `.grid()`.

Each window and `Frame` in your application can use only one geometry manager. However, different frames can use different geometry managers, even if they're assigned to a `Frame` or window using another geometry manager.

Let's start by taking a closer look at `.pack()`.

The `.pack()` Geometry Manager

`.pack()` uses a **packing algorithm** to place widgets in a `Frame` or window in a specified order. The packing algorithm has two primary steps:

1. It computes a rectangular area, called a **parcel**, that is just tall enough (or wide enough) to hold the widget, and it fills the remaining width (or height) in the window with blank space.
2. Unless you specify a different location, it centers the widget in the parcel.

.pack() is powerful, but it can be difficult to visualize. The best way to get a feel for .pack() is to look at some examples.

Let's see what happens when you .pack() three Label widgets into a Frame:

```
import tkinter as tk

window = tk.Tk()

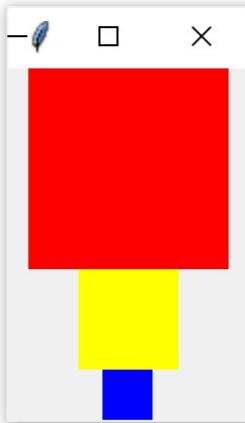
frame1 = tk.Frame(master=window, width=100, height=100, bg="red")
frame1.pack()

frame2 = tk.Frame(master=window, width=50, height=50, bg="yellow")
frame2.pack()

frame3 = tk.Frame(master=window, width=25, height=25, bg="blue")
frame3.pack()

window.mainloop()
```

By default, .pack() places each Frame below the previous one in the order they're assigned to the window:



Each `Frame` is placed at the topmost available position. The red `Frame` is placed at the top of the window. Then the yellow `Frame` is placed just below the red one, and the blue `Frame` just below the yellow one.

There are three invisible parcels, each containing one of the three `Frame` widgets. Each parcel is as wide as the window and as tall as the `Frame` that it contains. Since no anchor point was specified when `.pack()` was called for each `Frame`, they're all centered inside their parcels and are therefore centered in the window.

`.pack()` accepts some keyword arguments that allow you to configure your widget placement more precisely. For example, you can set the `fill` keyword argument to specify which direction the frames should fill. There are three options:

1. `tk.x` fills in the horizontal direction.
2. `tk.y` fills vertically.
3. `tk.BOTH` fills in both directions.

Here's how you would stack the three frames so that each one fills the whole window horizontally:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, height=100, bg="red")
frame1.pack(fill=tk.X)

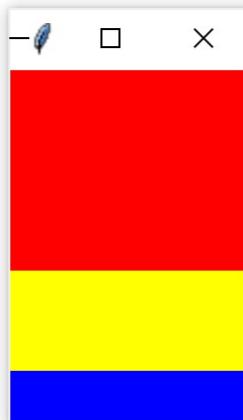
frame2 = tk.Frame(master=window, height=50, bg="yellow")
frame2.pack(fill=tk.X)

frame3 = tk.Frame(master=window, height=25, bg="blue")
frame3.pack(fill=tk.X)

window.mainloop()
```

Notice that the `width` isn't set for any of the `Frame` widgets. `width` is no longer necessary because `.pack()` is set to fill horizontally on each frame, overriding any width you may set.

The window produced by this code looks like this:



One of the nice things about filling the window with `.pack()` is that the fill is responsive to window resizing. Try widening the window generated by the previous code to see how this works.

As you widen the window, the width of the three `Frame` widgets grows to fill the window. Notice, though, that the `Frame` widgets don't expand in the vertical direction.

The `side` keyword argument of `.pack()` specifies on which side of the window the widget should be placed. The available options are `tk.TOP`, `tk.BOTTOM`, `tk.LEFT`, and `tk.RIGHT`. If you don't set `side`, then `.pack()` automatically uses `tk.TOP` and places new widgets at the top of the window or at the topmost portion of the window that isn't already occupied by a widget.

For example, the following program places three frames side by side from left to right and expands each frame to fill the window vertically:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, width=200, height=100, bg="red")
frame1.pack(fill=tk.Y, side=tk.LEFT)

frame2 = tk.Frame(master=window, width=100, bg="yellow")
frame2.pack(fill=tk.Y, side=tk.LEFT)

frame3 = tk.Frame(master=window, width=50, bg="blue")
frame3.pack(fill=tk.Y, side=tk.LEFT)

window.mainloop()
```

This time, you have to specify the `height` keyword argument on at least one of the frames to force the window to have some height.

The resulting window looks like this:



Just like setting `fill=tk.X` made the frames resize responsively when you resized the window horizontally, setting `fill=tk.Y` makes the frames resize responsively when you resize the window vertically. Try it out!

To make the layout truly responsive, you can set an initial size for your frames using the `width` and `height` attributes. Then set the `fill` keyword argument of `.pack()` to `tk.BOTH` and the `expand` keyword argument to `True`:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, width=200, height=100, bg="red")
frame1.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

frame2 = tk.Frame(master=window, width=100, bg="yellow")
frame2.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

frame3 = tk.Frame(master=window, width=50, bg="blue")
frame3.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

window.mainloop()
```

When you run the above code, you see a window that initially looks the same as the one generated in the previous example. The difference is that now you can resize the window however you want, and the frames will expand and fill the window responsively. Pretty cool!

The `.place()` Geometry Manager

You can use the `.place()` method of a widget to control the precise location that it should occupy in a window or `Frame`. You must provide two keyword arguments, `x` and `y`, that specify the `x`- and `y`-coordinates for the top-left corner of the widget. Both `x` and `y` are measured in pixels, not text units.

Keep in mind that the origin, where `x` and `y` are both 0, is the top-left corner of the `Frame` or window. You can think of the `y` argument of `.place()` as the number of pixels from the top edge of the window and the `x` argument as the number of pixels from the left edge.

Here's an example of how the `.place()` geometry manager works:

```
import tkinter as tk

window = tk.Tk()

# 1
frame = tk.Frame(master=window, width=150, height=150)
frame.pack()

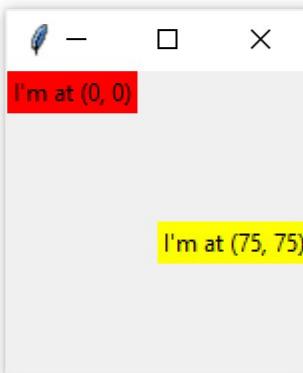
# 2
label1 = tk.Label(master=frame, text="I'm at (0, 0)", bg="red")
label1.place(x=0, y=0)

# 3
label2 = tk.Label(master=frame, text="I'm at (75, 75)", bg="yellow")
label2.place(x=75, y=75)

window.mainloop()
```

First, you create a new `Frame` widget called `frame` that's 150 pixels wide and 150 pixels tall and pack it into the window with `.pack()`. Then you create a `Label` with a red background called `label1` and place it in `frame1` at position (0, 0). Finally, you create a second `Label` with a yellow background called `label2` and place it in `frame1` at position (75, 75).

Here's the window that the code produces:



.place() isn't used often. It has two main drawbacks:

1. **Layouts can be difficult to manage with .place()**, especially if your application has lots of widgets.
2. **Layouts created with .place() are not responsive**. They don't change as the window is resized.

One of the main challenges of cross-platform GUI development is making layouts that look good no matter which platform they're viewed on. In most cases, .place() is a poor choice for making responsive and cross-platform layouts.

That's not to say .place() should never be used. It might be just what you need. For example, if you're creating a GUI interface for a map, then .place() might be the perfect choice to ensure widgets are placed at the correct distance from one another on the map.

.pack() is usually a better choice than .place(), but even .pack() has some downsides. For example, the placement of widgets depends on the order in which .pack() is called, so it can be difficult to modify existing applications without fully understanding the code controlling the layout.

As you'll see in the next section, the `.grid()` geometry manager solves a lot of these issues.

The `.grid()` Geometry Manager

The geometry manager you'll likely use most often is `.grid()`. It provides all the power of `.pack()` in a format that's easier to understand and maintain.

`.grid()` works by splitting a window or `Frame` into rows and columns. You specify the location of a widget by calling `.grid()` and passing the row and column indices to the `row` and `column` keyword arguments, respectively. Both row and column indices start at 0, so a row index of 1 and a column index of 2 tells `.grid()` to place a widget in the third column of the second row.

For example, the following code creates a 3×3 grid of frames with `Label` widgets packed into them:

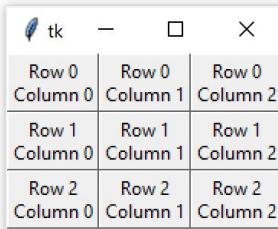
```
import tkinter as tk

window = tk.Tk()

for i in range(3):
    for j in range(3):
        frame = tk.Frame(
            master=window,
            relief=tk.RAISED,
            borderwidth=1
        )
        frame.grid(row=i, column=j)
        label = tk.Label(master=frame, text=f"Row {i}\nColumn {j}")
        label.pack()

window.mainloop()
```

Here's what the resulting window looks like:



Two geometry managers are used in this example. Each `Frame` is attached to the `window` with the `.grid()` geometry manager, and each `label` is attached to its master `Frame` with `.pack()`.

The important thing to realize here is that, even though `.grid()` is called on each `Frame` object, the geometry manager applies to the `window` object. Similarly, the layout of each `frame` is controlled by the `.pack()` geometry manager.

The frames in the previous example are placed tightly next to one another. To add some space around each `Frame`, you can set the padding of each cell in the grid. **Padding** is just blank space that surrounds a widget and visually separates it from its contents.

There are two types of padding: **external padding** and **internal padding**. External padding adds space around the outside of a grid cell. It's controlled with two keyword arguments of `.grid()`:

1. `padx` adds padding in the horizontal direction.
2. `pady` adds padding in the vertical direction.

Both `padx` and `pady` are measured in pixels, not text units, so setting both of them to the same value will create the same amount of padding in each direction.

Let's add some padding around the outside of the frames in the previous example:

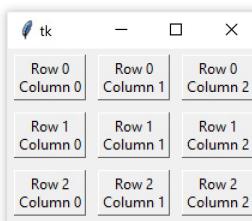
```
import tkinter as tk

window = tk.Tk()

for i in range(3):
    for j in range(3):
        frame = tk.Frame(
            master=window,
            relief=tk.RAISED,
            borderwidth=1
        )
        frame.grid(row=i, column=j, padx=5, pady=5)
        label = tk.Label(master=frame, text=f"Row {i}\nColumn {j}")
        label.pack()

window.mainloop()
```

Here's the resulting window:



.pack() also has `padx` and `pady` parameters. The following code is nearly identical to the previous code, except that 5 pixels of additional padding have been added around each `Label` in the both the `x` and `y` directions:

```
import tkinter as tk

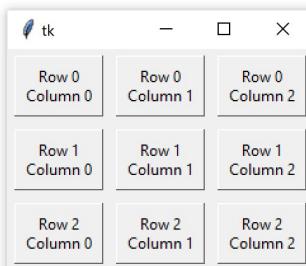
window = tk.Tk()

for i in range(3):
    for j in range(3):
        frame = tk.Frame(
            master=window,
            relief=tk.RAISED,
            borderwidth=1
        )
        frame.grid(row=i, column=j, padx=5, pady=5)

        label = tk.Label(master=frame, text=f"Row {i}\nColumn {j}")
        label.pack(padx=5, pady=5)

window.mainloop()
```

The extra padding around the `Label` widgets gives each cell in the grid a little bit of breathing room between the `Frame` border and the text in the `Label`:



That looks pretty nice! But if you try and expand the window in any direction, then you'll notice that the layout isn't very responsive. The whole grid stays in the top-left corner as the window expands.

You can adjust how the rows and columns of the grid grow as the window is resized using the `.columnconfigure()` and `.rowconfigure()` methods on the `window` object. Remember, the grid is attached to `window` even though you're calling `.grid()` on each `Frame` widget.

Both `.columnconfigure()` and `.rowconfigure()` take three essential arguments:

1. The index of the grid column or row that you want to configure (or a list of indices to configure multiple rows or columns)
2. A keyword argument called `weight` that determines how the column or row should respond to window resizing relative to the other columns and rows
3. A keyword argument called `minsize` that sets the minimum size of the row height or column width in pixels

`weight` is set to `0` by default, which means that the column or row doesn't expand as the window resizes. If every column and row is given a weight of `1`, then they all grow at the same rate. If one column has a weight of `1` and another has a weight of `2`, then the second column expands at twice the rate of the first.

Let's adjust the previous code to better handle window resizing:

```
import tkinter as tk

window = tk.Tk()

for i in range(3):
    window.columnconfigure(i, weight=1, minsize=75)
    window.rowconfigure(i, weight=1, minsize=50)

    for j in range(0, 3):
        frame = tk.Frame(
            master=window,
            relief=tk.RAISED,
            borderwidth=1
        )
        frame.grid(row=i, column=j, padx=5, pady=5)

        label = tk.Label(master=frame, text=f"Row {i}\nColumn {j}")
        label.pack(padx=5, pady=5)

window.mainloop()
```

The `.columnconfigure()` and `.rowconfigure()` methods are placed in the body of the outer `for` loop. You could explicitly configure each column and row outside of the `for` loop, but that would require writing an additional six lines of code.

On each iteration of the loop, the `i`th column and row are configured to have a `weight` of 1. This ensures that each row and column expands at the same rate whenever the window is resized.

The `minsize` argument is set to 75 for each column and 50 for each row. This makes sure the `Label` widget always displays its text without chopping off any characters, even if the window size is extremely small.

Try running the code to get a feel for how it works! Play around with the `weight` and `minsize` parameters to see how they affect the grid.

By default, widgets are centered in their grid cells. For example, the following code creates two `Label` widgets and places them in a grid with one column and two rows:

```
import tkinter as tk

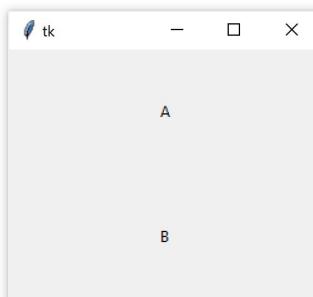
window = tk.Tk()
window.columnconfigure(0, minsize=250)
window.rowconfigure([0, 1], minsize=100)

label1 = tk.Label(text="A")
label1.grid(row=0, column=0)

label2 = tk.Label(text="B")
label2.grid(row=1, column=0)

window.mainloop()
```

Each grid cell is 250 pixels wide and 100 pixels tall. The labels are placed in the center of each cell, as you can see in the following figure:



You can change the location of each label inside the grid cell using the `.grid()` method's `sticky` parameter. `sticky` accepts a string containing one or more of the following letters:

- "`n`" or "`N`" to align to the top-center part of the cell
- "`s`" or "`S`" to align to the bottom-center part of the cell

- "e" or "E" to align to the right-center side of the cell
- "w" or "W" to align to the left-center side of the cell

The letters "n", "s", "e", and "w" come from the cardinal directions north, south, east, and west.

For example, setting `sticky` to "n" on both `Label` widgets in the previous code positions each `Label` at the top center of its grid cell:

```
import tkinter as tk

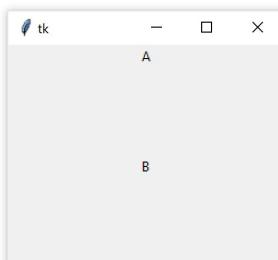
window = tk.Tk()
window.columnconfigure(0, minsize=250)
window.rowconfigure([0, 1], minsize=100)

label1 = tk.Label(text="A")
label1.grid(row=0, column=0, sticky="n")

label2 = tk.Label(text="B")
label2.grid(row=1, column=0, sticky="n")

window.mainloop()
```

Here's the output:



You can combine multiple letters in a single string to position each Label in a corner of its grid cell:

```
import tkinter as tk

window = tk.Tk()
window.columnconfigure(0, minsize=250)
window.rowconfigure([0, 1], minsize=100)

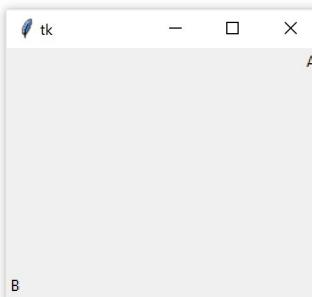
label1 = tk.Label(text="A")
label1.grid(row=0, column=0, sticky="ne")

label2 = tk.Label(text="B")
label2.grid(row=1, column=0, sticky="sw")

window.mainloop()
```

In this example, the `sticky` parameter of `label1` is set to "ne", which places the label at the top right corner of its grid cell. `label2` is positioned in the bottom left corner by passing "sw" to `sticky`.

Here's what that looks like in the window:



When you position a widget with `sticky`, the size of the widget is just big enough to contain the text and any other content inside the widget. It won't fill the entire grid cell.

To fill the grid, you can specify "ns", which forces the widget to fill the cell in the vertical direction, or "ew" to fill the cell in the horizontal direction. To fill the entire cell, set sticky to "nsew".

The following example illustrates each of these options:

```
import tkinter as tk

window = tk.Tk()

window.rowconfigure(0, minsize=50)
window.columnconfigure([0, 1, 2, 3], minsize=50)

label1 = tk.Label(text="1", bg="black", fg="white")
label2 = tk.Label(text="2", bg="black", fg="white")
label3 = tk.Label(text="3", bg="black", fg="white")
label4 = tk.Label(text="4", bg="black", fg="white")

label1.grid(row=0, column=0)
label2.grid(row=0, column=1, sticky="ew")
label3.grid(row=0, column=2, sticky="ns")
label4.grid(row=0, column=3, sticky="nsew")

window.mainloop()
```

Here's what the output looks like:



The above example illustrates that you can use the `.grid()` geometry manager's `sticky` parameter to achieve the same effects as the `.pack()` geometry manager's `fill` parameter.

The correspondence between the `sticky` and `fill` parameters is summarized in the following table:

<code>.grid()</code>	<code>.pack()</code>
<code>sticky="ns"</code>	<code>fill=tk.Y</code>
<code>sticky="ew"</code>	<code>fill=tk.X</code>
<code>sticky="nsew"</code>	<code>fill=tk.BOTH</code>

`.grid()` is a powerful geometry manager. It's often easier to understand than `.pack()` and is much more flexible than `.place()`. When creating new Tkinter applications, consider using `.grid()` as your primary geometry manager.

Note

`.grid()` offers much more flexibility than you've seen here. For example, you can configure cells to span multiple rows and columns.

For more information, check out the “[Grid Geometry Manager](#)” section of the [TkDocs tutorial](#).

Now that you have a handle on the basics of Tkinter’s geometry managers, the next step is to bring your applications to life by assigning actions to buttons.

Review Exercises

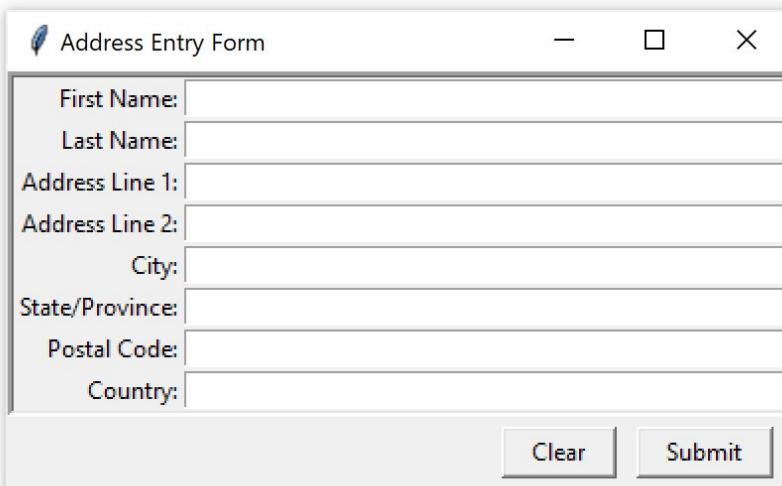
You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Try to re-create all the screenshots in this section without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for ten or fifteen minutes and try again.

Repeat this until you can produce all the screenshots on your own.

Focus on the output. It's okay if your own code is slightly different from the code in the book.

- Below is an image of a window made with Tkinter. Try to re-create the window using the techniques you've learned thus far. You may use any geometry manager you like.



18.7 Making Your Applications Interactive

By now, you have a pretty good idea of how to create a window with Tkinter, add some widgets, and control the application layout. That's great! But applications shouldn't just look good—they need to actually do something!

In this section, you'll learn how to bring your applications to life by performing actions whenever certain events occur.

Events and Event Handlers

When you create a Tkinter application, you must call `window.mainloop()` to start the **event loop**. During the event loop, your application checks if an event has occurred. If so, then it can execute some code in response.

The event loop is provided by Tkinter, so you don't have to write any code that checks for events. However, you *do* have to write the code that executes in response to an event. In Tkinter, you write functions called **event handlers** for the events that you use in your application.

So, what is an event, and what happens when one occurs?

An **event** is any action that occurs during the event loop, such as when the user presses a key or mouse button, that might trigger some behavior in the application.

When an event occurs, an **event object** is emitted, which means that an instance of a class representing the event is instantiated. You don't need to worry about creating these classes yourself. Tkinter will create instances of event classes for you automatically.

To better understand how Tkinter's event loop works, you can write your own event loop. That way, you can see how Tkinter's event loop fits into your application and which parts you need to write yourself.

Assume there's a list called `events_list` that contains event objects. Whenever an event occurs in your program, a new event object is appended to `events_list`. You don't need to implement this updating mechanism. It just magically happens for you in this make-believe example.

Using an infinite loop, you can continually check if any event objects are in `events_list`:

```
# Assume that this list gets updated automatically
events_list = []

# Run the event loop
while True:
    # If events_list is empty, then no events have occurred and you
    # can skip to the next iteration of the loop
    if events_list == []:
        continue

    # If execution reaches this point, then at least one
    # event object is in events_list
    event = events_list[0]
```

Right now, the event loop you've created doesn't do anything with `event`. Let's change that.

Suppose your application needs to respond to key presses. You need to check if `event` was generated by a user pressing a key on their keyboard and, if so, pass `event` to an event handler function for key presses.

You can assume that if the event is a keypress event object, then `event` has both a `.type` attribute set to the string "keypress" and a `.char` attribute containing the character of the key that was pressed.

Let's add a `handle_keypress()` function and update the event loop code:

```
events_list = []

# Create an event handler
def handle_keypress(event):
    """Print the character associated to the key pressed"""
    print(event.char)
```

```
while True:
    if events_list == []:
        continue
    event = events_list[0]

    # If event is a keypress event object
    if event.type == "keypress":
        # Call the keypress event handler
        handle_keypress(event)
```

When you call Tkinter's `window.mainloop()`, something like the above loop is run for you! Specifically, `.mainloop()` takes care of two parts of the loop for you:

1. It maintains a list of events that have occurred.
2. It runs an event handler any time a new event is added to the list.

You can update your event loop to use `window.mainloop()` instead of your own event loop:

```
import tkinter as tk

# Create a window object
window = tk.Tk()

# Create an event handler
def handle_keypress(event):
    """Print the character associated to the key pressed"""
    print(event.char)

# Run the event loop
window.mainloop()
```

`.mainloop()` takes care of a lot for you, but there's something missing from the above code. How exactly does Tkinter know when to use `handle_keypress()`?

The answer is that Tkinter widgets have a `.bind()` method that helps them do just that.

The `.bind()` Method

To call an event handler whenever an event occurs on a widget, you can use the widget's `.bind()` method. The event handler is said to be **bound** to the event because it's called every time the event occurs.

Continuing with the keypress example you saw in the previous section, you can use `.bind()` to bind `handle_keypress()` to the keypress event:

```
import tkinter as tk

window = tk.Tk()

def handle_keypress(event):
    """Print the character associated to the key pressed"""
    print(event.char)

# Bind keypress event to handle_keypress()
window.bind("<Key>", handle_keypress)

window.mainloop()
```

Here, you bind the `handle_keypress()` event handler to a "`<Key>`" event using `window.bind()`. Whenever a key is pressed while the application is running, the character of the key will be printed.

`.bind()` always takes two arguments:

1. An event represented by a string in the form "`<event_name>`", where `event_name` can be any of Tkinter's events
2. An event handler, which is the name of the function to be called whenever the event occurs

The event handler is bound to the widget on which `.bind()` is called.

When the event handler is called, the event object is passed to the event handler function.

In the above example, the event handler is bound to the window itself, but you can bind an event handler to any widget in your application.

For example, you can bind an event handler to a `Button` widget that will perform some action whenever the button is pressed:

```
def handle_click(event):
    print("The button was clicked!")

button = tk.Button(text="Click me!")

button.bind("<Button-1>", handle_click)
```

In this example, the "`<Button-1>`" event on the `button` widget is bound to the `handle_click` event handler. The "`<Button-1>`" event occurs whenever the left mouse button is pressed while the mouse is over the widget.

There are other events for mouse button clicks, including "`<Button-2>`" for the middle mouse button (if one exists) and "`<Button-3>`" for the right mouse button.

Note

For a list of commonly used events, see the “[Event Types](#)” section of the [Tkinter 8.5 reference](#).

You can bind any event handler to any kind of widget with `.bind()`, but there’s an easier way to bind event handlers to button clicks using the `Button` widget’s `command` attribute.

The `command` Attribute

Every `Button` widget has a `command` attribute that you can assign to a function. Whenever the button is pressed, the function is executed.

Let's look at an example. First, you'll create a window with a `Label` widget that holds a numerical value. You'll put a button on the left and right sides of the `Label`. The left button will be used to decrease the value in the `Label`, and the right one will increase the value.

Here's the code for the window:

```
import tkinter as tk

window = tk.Tk()

window.rowconfigure(0, minsize=50, weight=1)
window.columnconfigure([0, 1, 2], minsize=50, weight=1)

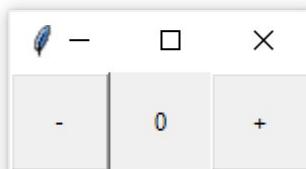
btn_decrease = tk.Button(master=window, text="-")
btn_decrease.grid(row=0, column=0, sticky="nsew")

lbl_value = tk.Label(master=window, text="0")
lbl_value.grid(row=0, column=1)

btn_increase = tk.Button(master=window, text="+")
btn_increase.grid(row=0, column=2, sticky="nsew")

window.mainloop()
```

The window looks like this:



With the app layout defined, you can bring it to life by giving the buttons some commands.

Let's start with the left button. When this button is pressed, it should decrease the value in the label by 1. To do this, there are two things you need to know how to do: how to *get* the text in a `Label` and how to *update* the text in a `Label`.

`Label` widgets don't have a `.get()` method like `Entry` and `Text` widgets do. However, you can retrieve the text from the label by accessing the `text` attribute with dictionary-style subscript notation:

```
label = Tk.Label(text="Hello")

# Retrieve a Label's text
text = label["text"]

# Set new text for the label
label["text"] = "Good bye"
```

Now that you know how to get and set a label's text, you can write a function that increases the value in the label by 1:

```
def increase():
    value = int(lbl_value["text"])
    lbl_value["text"] = f"{value + 1}"
```

`increase()` gets the text from `lbl_value` and converts it to an integer with `int()`. Then it increases this value by 1 and sets the label's `text` attribute to this new value.

You also need to write a `decrease()` function that decreases the value in `lbl_value` by 1:

```
def decrease():
    value = int(lbl_value["text"])
    lbl_value["text"] = f"{value - 1}"
```

Put the `increase()` and `decrease()` functions in your code just after the `import` statement.

To connect the buttons to the functions, assign the function to the button's `command` attribute. You can do this when you instantiate the button. For example, to assign `increase()` to `btn_increase`, update the line that instantiates the button to the following:

```
btn_increase = tk.Button(master=window, text="+", command=increase)
```

Now assign `decrease()` to `btn_decrease`:

```
btn_decrease = tk.Button(master=window, text="-", command=decrease)
```

That's all you need to do to bind the buttons to `increase()` and `decrease()` and make the program functional. Try saving your changes and running the application!

Here's the full application code for your reference:

```
import tkinter as tk

def increase():
    value = int(lbl_value["text"])
    lbl_value["text"] = f"{value + 1}"

def decrease():
    value = int(lbl_value["text"])
    lbl_value["text"] = f"{value - 1}"

window = tk.Tk()

window.rowconfigure(0, minsize=50, weight=1)
window.columnconfigure([0, 1, 2], minsize=50, weight=1)

btn_decrease = tk.Button(master=window, text="-", command=decrease)
btn_decrease.grid(row=0, column=0, sticky="nsew")

lbl_value = tk.Label(master=window, text="0")
```

```
lbl_value.grid(row=0, column=1)

btn_increase = tk.Button(master=window, text="+", command=increase)
btn_increase.grid(row=0, column=2, sticky="nsew")

window.mainloop()
```

This app is not particularly useful, but the skills you learned here apply to every app you'll make:

- Use **widgets** to create the **components** of the user interface.
- Use **geometry managers** to control the **layout** of the application.
- Write **functions** that interact with various components to capture and transform **user input**.

In the next two sections, you'll build apps that do something useful. First, you'll build a temperature converter that converts a temperature input from Fahrenheit to Celsius. After that, you'll build a text editor that can open, edit, and save text files!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

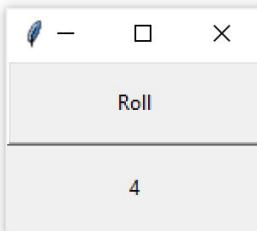
1. Write a program that displays a single button with the default background color and black text that reads "Click me".

When the user clicks the button, the button background should change to a color randomly selected from the following list:

```
["red", "orange", "yellow", "blue", "green", "indigo", "violet"]
```

2. Write a program that simulates rolling a six-sided die. There should be one button with the text "Roll". When the user clicks the button, a random integer from 1 to 6 should be displayed.

The application window should look something like this:



18.8 Example App: Temperature Converter

In this section, you'll build a temperature converter that allows the user to input a temperature in degrees Fahrenheit and push a button to convert that temperature to degrees Celsius.

We'll walk through the code step by step. You can also find the full source code at the end of this section for your reference.

To get the most out of this section, open up IDLE's editor window and follow along.

Before you start coding, take a moment to design the app. You need three basic elements:

1. An `Entry` widget called `ent_temperature` to enter the Fahrenheit value
2. A `Label` widget called `lbl_result` to display the Celsius result
3. A `Button` widget called `btn_convert` that reads the value from the `Entry` widget, converts it from Fahrenheit to Celsius, and sets the text of the `Label` widget to the result when clicked

You can arrange these in a grid with a single row and one column for each widget. That gets you a minimally working application, but it isn't very user-friendly. Everything needs to have some helpful labels.

Let's put a label containing the $^{\circ}\text{F}$ symbol directly to the right of the `ent_temperature` widget so that the user knows the value `ent_temperature` should be in degrees Fahrenheit. To do this, you'll set the label text to "`\N{DEGREE FAHRENHEIT}`", which uses Python's named Unicode character support to display the $^{\circ}\text{F}$ symbol.

You can give `btn_convert` a little flair by setting its text to the value "`\N{RIGHTWARDS BLACK ARROW}`", which displays a black arrow pointing to the right. You can also make sure that `lbl_result` always has the $^{\circ}\text{C}$ symbol by using "`\N{DEGREE CELSIUS}`" at the end to indicate that the result is in degrees Celsius.

Here's what the final window will look like:



Now that you know what widgets you need and what their window is going to look like, you're ready to start coding it up! First, import `tkinter` and create a new window:

```
import tkinter as tk

window = tk.Tk()
window.title("Temperature Converter")
```

`window.title()` sets the title of an existing window. When you finally run this application, the window will display "Temperature Converter" in its title bar.

Next, you'll create the `ent_temperature` widget along with a label called `lbl_temp` and assign both of them to a `Frame` widget called `frm_entry`:

```
frm_entry = tk.Frame(master=window)
ent_temperature = tk.Entry(master=frm_entry, width=10)
lbl_temp = tk.Label(master=frm_entry, text="\N{DEGREE FAHRENHEIT}")
```

`ent_temperature` is where the user will enter the Fahrenheit value, and `lbl_temp` will label `ent_temperature` with the °F symbol. `frm_entry` is just a container that groups `ent_temperature` and `lbl_temp` together.

You want `lbl_temp` to be placed directly to the right of `ent_temperature`, so you can lay them out in the `frm_entry` using the `.grid()` geometry manager with one row and two columns:

```
ent_temperature.grid(row=0, column=0, sticky="e")
lbl_temp.grid(row=0, column=1, sticky="w")
```

You set the `sticky` parameter of `ent_temperature` to "e" so that it always sticks to the rightmost edge of its grid cell. You also set the `sticky` parameter of `lbl_temp` to "w" to keep it stuck to the leftmost edge of its grid cell. These steps ensure that `lbl_temp` will always be located immediately to the right of `ent_temperature`.

Now you can make the `btn_convert` and the `lbl_result` for converting the temperature entered into `ent_temperature` and displaying the results:

```
btn_convert = tk.Button(
    master=window,
    text="\N{RIGHTWARDS BLACK ARROW}"
)
lbl_result = tk.Label(master=window, text="\N{DEGREE CELSIUS}")
```

Like `frm_entry`, both `btn_convert` and `lbl_result` are assigned to `window`. Together, these three widgets make up the three cells in the main application grid. Let's use `.grid()` to lay them out:

```
frm_entry.grid(row=0, column=0, padx=10)
btn_convert.grid(row=0, column=1, pady=10)
lbl_result.grid(row=0, column=2, padx=10)
```

Finally, run the application:

```
window.mainloop()
```

That looks great, but the button doesn't do anything yet. At the top of your code file, just below the `import` line, add a function called `fahrenheit_to_celsius()`. This function will read the user-supplied value from `ent_temperature`, convert it from Fahrenheit to Celsius, and then display the result in `lbl_result`:

```
def fahrenheit_to_celsius():
    """Convert the value from Fahrenheit to Celsius and insert the
    result into lbl_result.
    """
    fahrenheit = ent_temperature.get()
    celsius = (5/9) * (float(fahrenheit) - 32)
    lbl_result["text"] = f"{'round(celsius, 2)} \N{DEGREE CELSIUS}"
```

Now go down to the line where you define `btn_convert` and set its `command` parameter to `fahrenheit_to_celsius`:

```
btn_convert = tk.Button(
    master=window,
    text="\N{RIGHTWARDS BLACK ARROW}",
    command=fahrenheit_to_celsius  # <-- Add this line
)
```

That's it! You've created a fully functional temperature converter app in just twenty-six lines of code! Pretty cool, right?

Here's the full program for your reference:

```
import tkinter as tk

def fahrenheit_to_celsius():
    """Convert the value from Fahrenheit to Celsius and insert the
    result into lbl_result.

    """
    fahrenheit = ent_temperature.get()
    celsius = (5/9) * (float(fahrenheit) - 32)
    lbl_result["text"] = f"{round(celsius, 2)} \N{DEGREE CELSIUS}"

# Set up the window
window = tk.Tk()
window.title("Temperature Converter")
window.resizable(width=False, height=False)

# Create the Fahrenheit entry frame with an Entry
# widget and Label
frm_entry = tk.Frame(master=window)
ent_temperature = tk.Entry(master=frm_entry, width=10)
lbl_temp = tk.Label(master=frm_entry, text="\N{DEGREE FAHRENHEIT}")

# Lay out the temperature Entry and Label in frm_entry
# using the .grid() geometry manager
ent_temperature.grid(row=0, column=0, sticky="e")
lbl_temp.grid(row=0, column=1, sticky="w")

# Create the conversion Button and result display Label
btn_convert = tk.Button(
    master=window,
    text="\N{RIGHTWARDS BLACK ARROW}",
    command=fahrenheit_to_celsius
)
lbl_result = tk.Label(master=window, text="\N{DEGREE CELSIUS}")
```

```
# Set up the layout using the .grid() geometry manager
frm_entry.grid(row=0, column=0, padx=10)
btn_convert.grid(row=0, column=1, pady=10)
lbl_result.grid(row=0, column=2, padx=10)

# Run the application
window.mainloop()
```

Let's take things up a notch. Read on to build a simple text editor.

Review Exercise

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

1. Try to re-create the temperature converter app without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for ten or fifteen minutes and try again.

Repeat this until you can build the app from scratch on your own. Focus on the output. It's okay if your own code is slightly different from the code in the book.

18.9 Example App: Text Editor

In this section, you'll build a text editor app that can create, open, edit, and save text files.

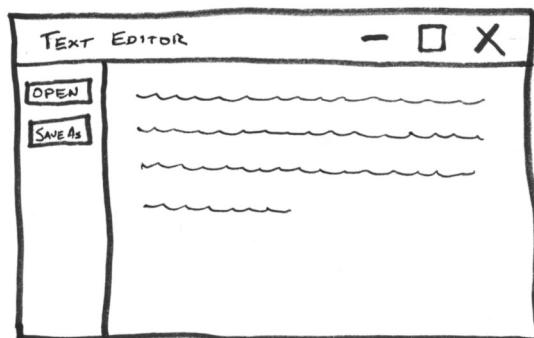
There are three essential elements in the application:

1. A `Button` widget called `btn_open` for opening a file for editing
2. A `Button` widget called `btn_save` for saving a file
3. A `TextBox` widget called `txt_edit` for creating and editing the text file

The three widgets will be arranged so that the two buttons are on the left side of the window, and the text box is on the right side.

The whole window should have a minimum height of 800 pixels, and `txt_edit` should have a minimum width of 800 pixels. The whole layout should be responsive so that if the window is resized, then `txt_edit` is resized as well. The width of the `Frame` holding the buttons should not change, however.

Here's a sketch of how the window will look:



You can achieve the desired layout using the `.grid()` geometry manager. The layout contains a single row and two columns: a narrow column on the left for the buttons and a wider column on the right for the text box.

To set the minimum sizes for the window and `txt_edit`, you can set the `minsize` parameters of the `.rowconfigure()` and `.columnconfigure()` window methods to 800. To handle resizing, you can set the `weight` parameters of these methods to 1.

To get both buttons into the same column, you'll need to create a `Frame` widget, which you can call `fr_buttons`. According to the sketch, the two buttons should be stacked vertically inside this frame, with `btn_open` on top. You can do that with either the `.grid()` or the `.pack()` geometry manager, but let's stick with `.grid()` since it's a little less complicated to work with.

Now that you have a plan, you can start coding the application. The first step is to create all the widgets you need:

```
import tkinter as tk

# 1
window = tk.Tk()
window.title("Simple Text Editor")

# 2
window.rowconfigure(0, minsize=800, weight=1)
window.columnconfigure(1, minsize=800, weight=1)

# 3
txt_edit = tk.Text(window)
fr_buttons = tk.Frame(window)
btn_open = tk.Button(fr_buttons, text="Open")
btn_save = tk.Button(fr_buttons, text="Save As...")
```

First (#1), you import `tkinter` and create a new window with the title "Simple Text Editor". Then (#2) you set the row and column configurations . Finally (#3), you create four widgets: the `txt_edit` text box, the `fr_buttons` frame, and the `btn_open` and `btn_save` buttons.

Take a closer look at #2. The `minsize` parameter of `.rowconfigure()` is set to 800 and `weight` is set to 1. The first argument is 0, so this sets the height of the first row to 800 pixels and makes sure that the height of the row grows proportionally to the height of the window. There's only one row in the application layout, so these settings apply to the entire window.

On the next line, `.columnconfigure()` is used to set the `width` and `weight` attributes of the column with index 1 to 800 and 1, respectively. Remember, row and column indices are zero-based, so these settings apply only to the second column.

By configuring just the second column, you ensure that the text box will expand and contract naturally when the window is resized, while the column containing the buttons will remain at a fixed width.

Now you can work on the application layout. First, you'll assign the two buttons to the `fr_buttons` frame using the `.grid()` geometry manager:

```
btn_open.grid(row=0, column=0, sticky="ew", padx=5, pady=5)
btn_save.grid(row=1, column=0, sticky="ew", padx=5)
```

These two lines of code create a grid with two rows and one column in the `fr_buttons` frame, since both `btn_open` and `btn_save` have their `master` attribute set to `fr_buttons`. `btn_open` is put in the first row and `btn_save` in the second row so that `btn_open` appears above `btn_save` in the layout, just as you planned in your sketch.

Both `btn_open` and `btn_save` have their `sticky` attributes set to "ew", which forces the buttons to expand horizontally in both directions and fill the entire frame. This makes sure both buttons are the same size.

You place 5 pixels of padding around each button by setting the `padx` and `pady` parameters to 5. Only `btn_open` has vertical padding. Since `btn_open` is on top, the vertical padding offsets the button down from the top of the window a bit and makes sure that there's a small gap between `btn_open` and `btn_save`.

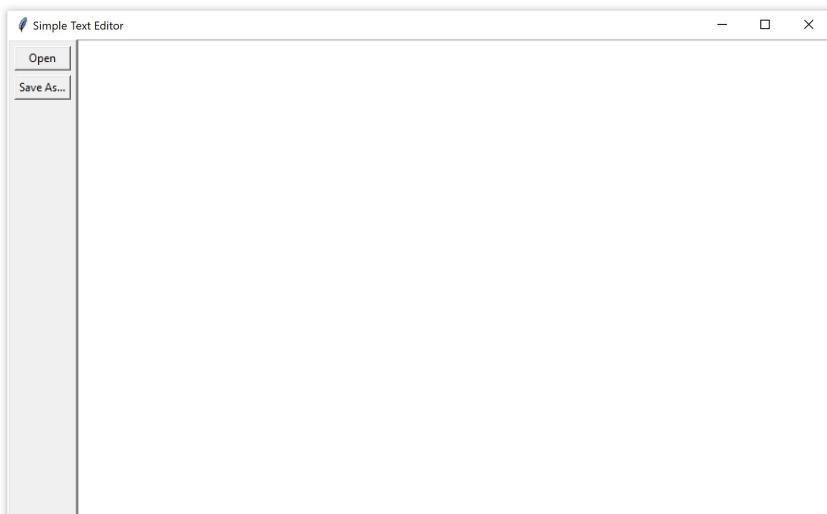
Now that `fr_buttons` is laid out and ready to go, you can set up the grid layout for the rest of the window:

```
fr_buttons.grid(row=0, column=0, sticky="ns")
txt_edit.grid(row=0, column=1, sticky="nsew")
```

These two lines of code create a grid with one row and two columns for `window`. `fr_buttons` is placed in the first column and `txt_edit` in the second column so that `fr_buttons` appears to the left of `txt_edit` in the window layout.

The `sticky` parameter for `fr_buttons` is set to "ns", which forces the whole frame to expand vertically and fill the entire height of its column. `txt_edit` fills its entire grid cell because its `sticky` parameter is set to "nsew", which forces it to expand in every direction.

Now that the application layout is complete, add `window.mainloop()` to the bottom of the program, then save and run the file. The following window is displayed:



That looks great! But it doesn't do anything yet, so let's start writing the commands for the buttons.

The `btn_open` button needs to show a file open dialog and allow the user to select a file. Then it needs to open that file and set the text of `txt_edit` to the contents of the file.

Here's a function, `open_file()`, that does just this:

```
def open_file():
    """Open a file for editing."""
    # 1
    filepath = askopenfilename(
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]
    )

    # 2
    if not filepath:
        return

    # 3
    txt_edit.delete("1.0", tk.END)

    # 4
    with open(filepath, "r") as input_file:
        text = input_file.read()
        txt_edit.insert(tk.END, text)

    # 5
    window.title(f"Simple Text Editor - {filepath}")
```

First (#1), you use the `askopenfilename` dialog from the `tkinter.filedialog` module to display a file open dialog and store the selected file path to the `filepath` variable. If the user closes the dialog box or clicks `Cancel` (#2), then `filepath` is `None` and the function returns without executing any of the code to read the file and set the text of `txt_edit`.

If the user does choose a file (#3), then the current contents of `txt_edit` are cleared using `.delete()`. Then (#4) the select file is opened and the contents of the file are read using `.read()` and stored as a string in the `text` variable. The string `text` is assigned to `txt_edit` using `.insert()`.

Finally (#5), the title of the window is set so that it contains the path of the open file.

Now you can update the program so that `btn_open` calls `open_file()` whenever it's clicked. There are three things you need to do to update the program:

1. Import `askopenfilename()` from `tkinter.filedialog` by adding the following import to the top of your program:

```
from tkinter.filedialog import askopenfilename
```

2. Add the definition of `open_file()` just below the import statements.
3. Set the `command` attribute of `btn_open` to `open_file`:

```
btn_open = tk.Button(fr_buttons, text="Open", command=open_file)
```

Save the file and run it to check that everything is working. Try opening a text file!

Note

If you have trouble getting the updates to work, then you can skip ahead to the end of this section to see the full code for the text editor application.

With `btn_open` working, let's work on the function for `btn_save`. It needs to open a save file dialog box so that the user can choose where they would like to save the file. You'll use the `asksaveasfilename` dialog in the `tkinter.filedialog` module for this. It also needs to extract the text currently in `txt_edit` and write this to a file at the selected location.

Here's a function that does just this:

```
def save_file():
    """Save the current file as a new file."""
    # 1
    filepath = asksaveasfilename(
        defaultextension=".txt",
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")],
```

```
)  
  
# 2  
if not filepath:  
    return  
  
# 3  
with open(filepath, "w") as output_file:  
    text = txt_edit.get("1.0", tk.END)  
    output_file.write(text)  
  
# 4  
window.title(f"Simple Text Editor - {filepath}")
```

First (#1), the `asksaveasfilename` dialog box gets the desired save location from the user, and the selected file path is stored in the `filepath` variable. If the user closes the dialog box or clicks `Cancel` (#2), then `filepath` is `None` and the function returns without executing any of the code to save the text to a file.

If the user does select a file path (#3), then a new file is created. The text from `txt_edit` is extracted with the `.get()` method and assigned to the variable `text` and written to the output file.

Finally (#4), the title of the window is updated so that the new file path is displayed in the window title.

Now you can update the program so that `btn_save` calls `save_file()` when it's clicked. There are three things you need to do in order to update the program:

1. Import the `asksaveasfilename()` function from `tkinter.filedialog` by updating the import at the top of your program, like so:

```
from tkinter.filedialog import askopenfilename, asksaveasfilename
```

2. Add the definition of `save_file()` just below the `open_file()` definition.

- Set the `command` attribute of `btn_save` to `save_file`:

```
btn_save = tk.Button(  
    fr_buttons, text="Save As...", command=save_file  
)
```

Save the file and run it. You now have a minimal yet fully functional text editor!

Here's the full program for your reference:

```
import tkinter as tk  
from tkinter.filedialog import askopenfilename, asksaveasfilename  
  
def open_file():  
    """Open a file for editing."""  
    filepath = askopenfilename(  
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")]  
    )  
    if not filepath:  
        return  
    txt_edit.delete(1.0, tk.END)  
    with open(filepath, "r") as input_file:  
        text = input_file.read()  
        txt_edit.insert(tk.END, text)  
    window.title(f"Simple Text Editor - {filepath}")  
  
def save_file():  
    """Save the current file as a new file."""  
    filepath = asksaveasfilename(  
        defaultextension=".txt",  
        filetypes=[("Text Files", "*.txt"), ("All Files", "*.*")],  
    )  
    if not filepath:  
        return  
    with open(filepath, "w") as output_file:  
        text = txt_edit.get(1.0, tk.END)
```

```
        output_file.write(text)
    window.title(f"Simple Text Editor - {filepath}")

window = tk.Tk()
window.title("Simple Text Editor")
window.rowconfigure(0, minsize=800, weight=1)
window.columnconfigure(1, minsize=800, weight=1)

txt_edit = tk.Text(window)
fr_buttons = tk.Frame(window, relief=tk.RAISED, bd=2)
btn_open = tk.Button(fr_buttons, text="Open", command=open_file)
btn_save = tk.Button(fr_buttons, text="Save As...", command=save_file)

btn_open.grid(row=0, column=0, sticky="ew", padx=5, pady=5)
btn_save.grid(row=1, column=0, sticky="ew", padx=5)

fr_buttons.grid(row=0, column=0, sticky="ns")
txt_edit.grid(row=0, column=1, sticky="nsew")

window.mainloop()
```

You've now built two GUI applications in Python. In doing so, you've applied many of the topics you've learned about throughout this book. That's no small achievement, so take some time to feel good about what you've done!

You're now ready to tackle some applications on your own!

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources

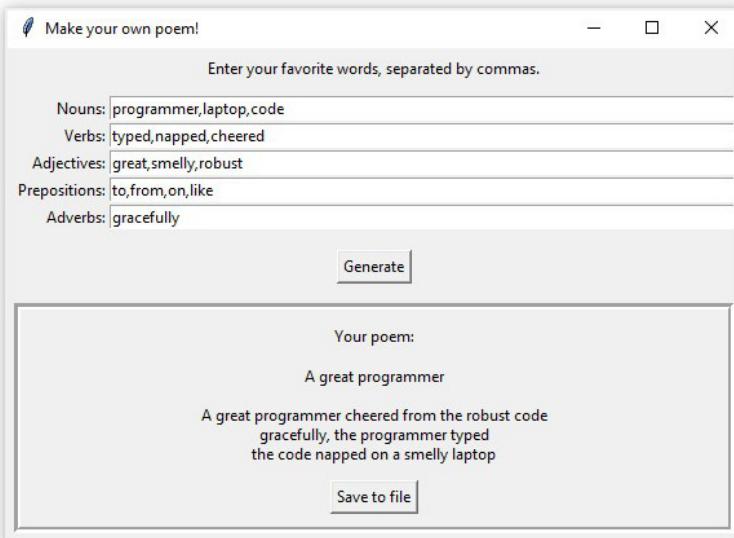
1. Try to re-create the text editor app without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for ten or fifteen minutes and try again.

Repeat this until you can build the application from scratch on your own. Focus on the output. It's okay if your own code is slightly different from the code in the book.

18.10 Challenge: Return of the Poet

For this challenge, you'll write a GUI application for generating poetry. This application is based on the poem generator from chapter 9.

Visually, the application should look similar to this:



You may use whichever geometry manager you like, but the application should do all of the following:

1. The user should be required to enter the correct number of words in each `Entry` widget:
 - At least three nouns
 - At least three verbs
 - At least three adjectives

- At least three prepositions
- At least one adverb

If too few words are entered into any of the `Entry` widgets, then an error message should be displayed in the area where the generated poem is shown.

2. The program should randomly choose three nouns, three adverbs, three adjectives, and three prepositions as well as one adverb from the user input.
3. The program should generate the poem using the following template:

```
{A/An} {adj1} {noun1}
```

```
A {adj1} {noun1} {verb1} {prep1} the {adj2} {noun2}  
{adverb1}, the {noun1} {verb2}  
the {noun2} {verb3} {prep2} a {adj3} {noun3}
```

4. The application must allow the user to export their poem to a file.
5. *Bonus:* Check that the user inputs unique words into each `Entry` widget. For example, if the user enters the same noun into the noun `Entry` widget twice, then the application should display an error message when the user tries to generate the poem.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources

18.11 Summary and Additional Resources

In this chapter, you learned how to build some basic graphical user interfaces (GUIs).

First, you learned how to use the EasyGUI package to create dialog boxes that display messages, accept user input, and allow a user to select files for reading and writing. Then you learned about Tkinter, which is Python’s built-in GUI framework. Tkinter is more complex than EasyGUI, but it’s also more flexible.

You learned how to work with widgets in Tkinter, including `Frame`, `Label`, `Button`, `Entry`, and `Text` widgets. Widgets can be customized by assigning values to their various attributes. For example, setting the `text` attribute of a `Label` widget assigns some text to the label.

Next, you saw how to use Tkinter’s `.pack()`, `.place()`, and `.grid()` geometry managers to give your GUI applications a layout. You learned how to control various aspects of the layout, including internal and external padding, and how to create responsive layouts with the `.pack()` and `.grid()` managers.

Finally, you brought all of these skills together to create two full GUI applications: a temperature converter and a simple text editor.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/pybasics-gui

Additional Resources

To learn more about GUI programming in Python, check out these resources:

- [Tkinter tutorial](#)

For links and additional resources to further deepen your Python skills, visit realpython.com/python-basics/resources

Chapter 19

Final Thoughts and Next Steps

Congratulations! You've made it to all the way to the end of this book. You already know enough to do a lot of amazing things with Python, but now the real fun starts: it's time to explore on your own!

The best way to learn is by solving real problems that you face in everyday life. Sure, your code might not be very pretty or efficient when you're just starting out, but it will be useful. Need some inspiration? Check out "[13 Project Ideas for Intermediate Python Developers](#)" for some ideas to get you started!

Part of what makes Python so great is the community. Know someone else learning Python? Help them out! The only way to really master a concept is to explain it to someone else.

Next up, dive into the more advanced material available at [realpython.com](#) or peruse the articles and tutorials featured in the [PyCoder's Weekly](#) newsletter.

When you feel ready, consider helping out with an open source project on [GitHub](#). If puzzles are more your style, try working through some of the mathematical challenges on [Project Euler](#).

If you get stuck somewhere along the way, it’s almost guaranteed that someone else has encountered (and potentially solved) the exact same problem before. Search around for answers, particularly at [Stack Overflow](#), or find a [community](#) of Pythonistas willing to help out.

If all else fails, `import this` and take a moment to meditate on that which is Python.

P.S. Come visit us on the Web and continue your Python journey on the [realpython.com](#) website and the [@realpython](#) Twitter account.

19.1 Free Weekly Tips for Python Developers

Are you looking for a weekly dose of Python development tips to improve your productivity and streamline your workflows? Good news—we’re running a free email newsletter for Python developers just like you.

The newsletter emails we send out are not your typical “here’s a list of popular articles” flavor. Instead, we aim to share at least one original thought per week in a (short) essay-style format.

If you’d like to see what all the fuss is about, then head on over to [realpython.com/newsletter](#) and enter your email address in the signup form. We’re looking forward to meeting you!

19.2 Python Tricks: The Book

Now that you’re familiar with the basics of Python, it’s time to dig in deeper and round out your knowledge.

With *Real Python’s Python Tricks* book, you’ll discover Python’s best practices and the power of beautiful, Pythonic code with simple examples and a step-by-step narrative.

You'll get one step closer to mastering Python so that you can write beautiful and idiomatic code that comes to you naturally.

Learning the ins and outs of Python is difficult, and with this book you'll be able to focus on taking your core Python skills to the next level.

Discover the hidden treasures in Python's standard library and start writing clean and Pythonic code today. Download a free sample chapter at realpython.com/pytricks-book

19.3 Real Python Video Course Library

Become a well-rounded Pythonista with *Real Python*'s large (and growing) collection of Python tutorials and in-depth training materials. With new content published weekly, you'll always find something to boost your skills:

- **Master practical, real-world Python skills:** Our tutorials are created, curated, and vetted by a community of expert Pythonistas. At Real Python, you'll get the trusted resources you need on your path to Python mastery.
- **Meet other Pythonistas:** Join the *Real Python* Community chat and weekly Office Hours Q&A calls to meet the *Real Python* team along with other learners. Get your Python questions answered, discuss coding and career topics, or just hang out with us at this virtual water cooler.
- **Explore interactive quizzes and learning paths:** See where you stand and practice what you learn with interactive quizzes, hands-on coding challenges, and skills-focused learning paths.
- **Track your learning progress:** Mark lessons as completed or in-progress and learn at your own comfortable pace. Bookmark interesting lessons and review them later to boost long-term retention.

- **Earn completion certificates:** For each course you complete, you'll receive a shareable (and printable) certificate of completion. Embed your certificates in your portfolio, LinkedIn resume, and other websites to show the world that you're a dedicated Pythonista.
- **Stay up to date:** Keep your skills fresh and keep up with technology. We're constantly releasing new members-only tutorials and we update our content regularly.

See what's available at realpython.com/courses

19.4 Acknowledgements

This book would not have been possible without the help and support of so many friends and colleagues. We would like to thank many people for their assistance in making this book possible:

- **Our families:** Thank you for bearing with us through crunch mode as we worked night and day to get this book into readers' hands.
- **The CPython team:** Thank you for producing the amazing programming language and tools that we love and work with every day.
- **The Python community:** Thank you for working hard to make Python the most beginner-friendly and welcoming programming language in the world, for running conferences, and for maintaining critical infrastructure like PyPI.
- **The readers of realpython.com, like you:** Thank you so much for reading our online articles and purchasing this book. Your continued support and readership is what makes all of this possible!

We hope that you will continue to be active in the community, asking questions and sharing tips. Reader feedback has shaped this book over the years and will continue to help us make improvements in future editions, so we look forward to hearing from you.

Our deepest thanks to all the Kickstarter backers who took a chance on this project in 2012. We never expected to gather such a large group of helpful, encouraging people.

Finally, we would like to thank our early access readers for their excellent feedback: Zoheb Ainapore, Luther Reed, Rob Sandusky, Luther, Marc, Ricky Mitchell, Robert Livingston, Wayne, Tom Moens, Meir Guttman, Larry Eisenberg, Ricky, Phu Le, Jeffrey Hansen, Albrecht, Mark Palie, Peter Aronoff, Kilimandaros, Patricio Urrutia, Joanna Jablonski, Miguel Alves, Mursalin Simpson, Xu Chunyang, Lucas, Ward Walker, W., Vlad, Jim Anderson, Mohamed Alshishani, Melvin, Albrecht Kadauke, Patrick Starrenburg, Vivek, Srinivasan Samuel, Sampath, Ceejay Cervantes, Liam, Ty Wait, Marp, Jorge Alberch, Edythe, Miguel Galán, Tom Carnevale, Florent, Peter, Jon Radue, Matt Gardner, Robert, Sean Yang, David S., Hans van Nielen, Youri Torchalski, Gavin, Karen H Calhoun MD, Roman, Robert Robb Livingston, Terence Phillips, Nico, Daniel, W, Cairo DeGaillard, Lucas das Dores, David, Dave, Tony Denning, Sean, Peter Kronfeld, Mark, Dennis Miller, Joseph Araneta Jr., Nathan Eger, Kumaran Rajendhiran, David Fullerton, Nicklas, Jacob Andersen, Mario, Alejandro Ramos, Beni_begin, AJ, Don Edwards, Jon, Ridwan Mizan, Graham Kneen, Iliyan, Helmut, Izak Zycer, Mike, Norman Greenwood, Forrest, Patricio, Rene, Richard Mertz, Chris Robinson, Pete Storer, Russ Garside, Matt, Richard, Russ Garside, Tiago Mendes, Michael, Daniel Alves Mertins, Marko Umek, Chris Jenks, Eddy, Dmitry, Kelsang Sherab, Thomas, Dom Jennings, Martin, Anthony Sheffield, S F, Velu V, Peter Cavallaro, Charlie Browning 3, Milind Mahajani, Jason Barnes, Lucien Boland, Adam Bretel, William, Veltaine, Jerry Petrey, James, Raymond E Rogers, Ty Wait, Bimperng Uen, CJ Hwang, Guido, Evan, Miguel Galan, Han Qi, Jim Bremner, Matt Chang, Daniel Drazan, Cole, Bob, Reed Howald, Edward Duarte, Mike Parker, Aart Kleinendorst, Rock,

Johnny, Rock Lee, Dusan Ranisavljev, Grant, Jack, Reinhard, Vivek Vashist, Dan, Garrett, Jun Lee, James Silk, Nik Singhal, Charles, Allard Schmidt, Jeff Desalle, Miguel, Steve Poe, Jonathan Seubert, Marc Poulin, Lee Jordan, Matthew Chin, James Mitchell, Wayne, Zarata, Lisa, Ryan Otero, Lee, Raphael Bytebier, Graeme Edwards, Jeff Skipper, Bob D, Anderson Tomazeli, Selemani Said Jawa, Meow Carter, Russ Garside, Louis Sheldon, James Radford, Nikkolai Jones, George Zagas, Len Gould, Daniel Kapitan, Chris, Sheng Jun, Walt Busse, Melissa Gregoire, Mohammad Nassar, Carles Casademunt, Forrest Smith, Aurel Weisswange, Russ, Wolfram Blechner, Tony Denning, Ron Fenimore, Edward Wright, Justin, Darren Olive, Charlie Clemmer, Dwayne Reid, Waiman Yau, C. Scott Kippen, Jimmy, Wolfram Blechner, Mark Mathewson, François iBrunet, Jeff Cabral, Bjorn, Jason Williams, Scott Page, Marilyn Gartley, Lief Rutzebeck, Mustafa Adaoglu, Thejan, Thejan Rathnayake, Cindy Ancrum, Tati Carvalho, Marek Ratiborsky, Ben, Francis Adepoju, Nir, Prabhu, Steve Fisher, Carlos, Aaron, David Maietta, Michael Huckleberry, Pawel, Julio Cesar Zebadua, Vencislav Shoykov, Michael Klengel, Kerry Alfred, Afeez Popoola, Cindy A., LC, tfig, Tiago, Sophie Wang, Toshiko, Fahmi, Paul Pennington, Wer, Jeff Johnson, Dutchy, Cesar, Albrecht KAdauke, Jim Brown, Eric, Christopher Evans, MELVIN, Idris, John Chirico, Wynette Espinosa, J.P., Gregory, Mark Edgeller, David Melanson, Raul Pena, Darrell, Shriram, Tom Flynn, Velu, Michael Lindsey, Sulo Kolehmainen, Jay, Milos “Ozzyx” Kosik, Hans de Cocq, Glen Mules, Nathan Lundner, Phil, Shubh, Puwei Wang, Alex Mück, Alex, Hitoshi, Bruno F. De Lima, Dario David, Rajesh, Haroldas Valčiukas, GVeltaine, Susan Fowle, Jared Simms, Nathan Collins, Dylan, Les Churchman, Stephane Li-Thiao-Te, Frank P, Paul, Damien Murtagh, Jason, Thắng Lê Quang, Neill, Lele, Charles Wilson, Damien, Christian, Andreas Kreisig, Marco, Mario Panagiotopoulos, Nerino, Mariusz, Mihhail, Mikönig, Fabio, Scott, A, Pedro Torres, Mathias Johansson, Joshua S., Mathias, Scott, David Koppy, Rohit Bharti, Phillip Douglas, John Stephenson, Jeff Jones, George Mast, Allards, Palak, Nikola N., Palak Kalsi, Annekathrin, Tsung-Ju Yang, Nick Huntington, Sai, Jordan, Wim Alsemgeest, DJ, Bob Harris, Andrew, Reggie Smith, Steve Santy, Mohee Jarada, Mark Arzaga, Poulose Matthen, Brent Gordon, Gary Butler, Bryant,

Dana, Koajck, Reggie, Luis Bravo, Elijah, Nikolay, Eric Lietsch, Fred Janssen, Don Stillwell, Gaurav Sharma, Mike McKenna, Karthik Babu, Bulat Mansurov, August Trillanes, Darren Saw, Jagadish, Kyle, Tejas Shetty, Baba Sariffodeen, Don, Ian, Ian Barbour, Redhouane, Wayne Rosing, Emanuel, Toigongonbai, Jason Castillo, Krishna Chaitanya Swamy Kesavarapu, Corey Huguley, Nick, Xuchunyang, Daniel Buis, Kenneth, Leodanis Pozo Ramos, John Phenix, Linda Moran, W Laleau, Troy Flynn, Heber Nielsen, Rock, Mike LeRoy, Thomas Davis, Jacob, Szabolcs Sinka, Kalaiselvan, Leanne Kuss, Andrey, Omar, Jason Woden, David Cebalo, John Miller, David Bui, Nico Zanferrari, Ariel, Boris, Boris Ender, Charlie3, Ossy, Matthias Kuehl, Scott Koch, Jesus Avina, Charlie, Awadhesh, Andie, Chris Johnson, Malan, Ciro, Thamizhselvan, Neha, Christian Langpap, Ivan, Dr. Craig Levy, H B Robinson, Stéphane, Steve McIlree, Yves, Teresa, Allard, Tom Cone Jr., Dirk, Joachim van der Weijden, Jim Woodward, Christoph Lipka, John Vergelli, Gerry, Lu, Robert R., Vlad, Richard Heatwole, Gabriel, Krzysztof Surowiecki, Alexandra Davis, Jason Voll, and Dwayne Dever.

If we've forgotten to mention your name here, then please know we're extremely grateful for your help. Thank you all!