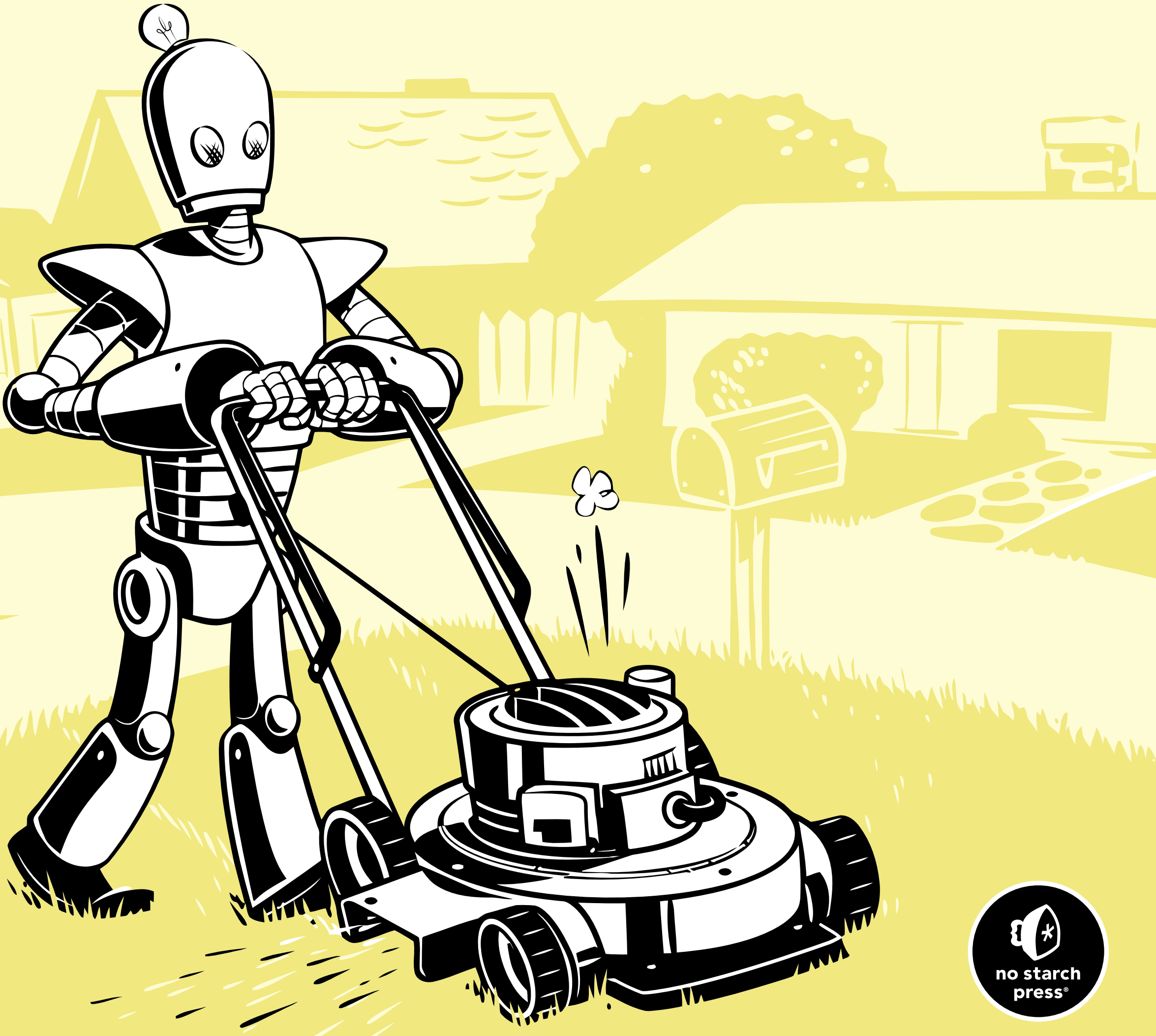
EARLY ACCESS

THIRD EDITION

AUTOMATE THE BORING STUFF WITH PYTHON

PRACTICAL PROGRAMMING FOR TOTAL BEGINNERS

AL SWEIGART



no starch press

NO STARCH PRESS EARLY ACCESS PROGRAM: FEEDBACK WELCOME!

Welcome to the Early Access edition of the as yet unpublished *Automate the Boring Stuff with Python, 3rd edition* by Al Sweigart! As a prepublication title, this book may be incomplete and some chapters may not have been proofread.

Our goal is always to make the best books possible, and we look forward to hearing your thoughts. If you have any comments or questions, email us at **earlyaccess@nostarch.com**. If you have specific feedback for us, please include the page number, book title, and edition date in your note, and we'll be sure to review it. We appreciate your help and support!

We'll email you as new chapters become available. In the meantime, enjoy!

AUTOMATE THE BORING STUFF WITH PYTHON, 3RD EDITION

AL SWEIGART

Early Access edition, 11/08/2024

Copyright © 2025 by Al Sweigart.

ISBN 13: 978-1-7185-0340-3 (print)
ISBN 13: 978-1-7185-0341-0 (ebook)

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Allison Felus
Developmental Editor: Frances Saux
Cover Illustrator: Josh Ellingson
Interior Design: Octopod Studios
Technical Reviewer: Daniel Zingaro
Copyeditor: Audrey Doyle

No Starch Press and the No Starch Press logo are registered trademarks of No Starch Press, Inc. Other product and company names mentioned herein may be the trademarks of their respective owners. Rather than use a trademark symbol with every occurrence of a trademarked name, we are using the names only in an editorial fashion and to the benefit of the trademark owner, with no intention of infringement of the trademark.

All rights reserved. No part of this work may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the copyright owner and the publisher.

The information in this book is distributed on an “As Is” basis, without warranty. While every precaution has been taken in the preparation of this work, neither the author nor No Starch Press, Inc. shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in it.

CONTENTS

The chapters in **red** are included in this Early Access PDF.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

FOREWORD

I’ve known Al Sweigart for as long as his books have been in print. I first knew Al as a reader, then as a colleague, and more recently as a friend. Al thinks deeply and critically about programming, how people learn, and how we live together as a society. You couldn’t ask for a better person to learn from.

There are two main reasons people tend to want to learn about programming. Many have a particular task to automate or a specific problem they want to solve using code, while others have a more general interest in learning a new skill. Often, people are motivated to learn for a mix of these two reasons. In any case, Python is a perfect language to study. It has a wide variety of libraries you can download to help you automate almost any task you can think of, and you’ll find a large number of Python resources that will almost certainly let you get to work quickly solving the problems you care about.

Automate the Boring Stuff with Python was a brilliant concept for a book when it first came out 10 years ago, and it remains a brilliant concept today. This is evidenced by the fact that it’s been one of the go-to resources for learning Python for as long as it’s been in print. If you need to solve a specific task right now, you’ll find many practical examples in this book that you can adapt to your own needs. If you’re interested in developing a more general understanding of programming, implementing a series of real-world projects like the ones you’ll find here is a great way to do so.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

There has never been a better time to learn Python. Some people, believing the hype about artificial intelligence, will say that you don't need to learn programming anymore because AI tools can write all the code for you. Others will say that AI-generated code is terrible and will never work. The reality, as is almost always the case, lies somewhere between these two extremes.

AI tools can certainly help you in your programming work. But they work much better if you already have a reasonable understanding of programming in general and know how you can use programming to solve the specific task you're working on. Otherwise, you'll almost certainly run into a sticking point that neither you nor the AI assistant can get past. If you use an AI assistant with the knowledge and understanding you gain from this book, however, you'll be able to build useful and effective tools for yourself quite efficiently.

Enjoy the journey; it's a great one!

Eric Matthes
Author of *Python Crash Course,* 3rd Edition (No Starch Press, 2023)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

ACKNOWLEDGMENTS

It’s misleading to have just my name on the cover.

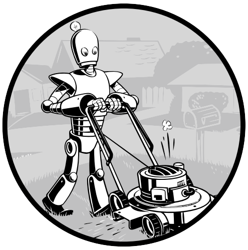
I couldn’t have written a book like this without the help of a lot of people. I’d like to thank my publisher, Bill Pollock; my editors, Laurel Chun, Leslie Shen, Greg Poulos, Jennifer Griffith-Delgado, Frances Saux, Jill Franklin, Sabrina Plomitallo-González, and Allison Felus; and the rest of the staff at No Starch Press for their invaluable help. Thanks to my tech reviewers, Ari Lacenski, Philip James, and Dr. Daniel Zingaro, for great suggestions, edits, and support.

Many thanks to everyone at the Python Software Foundation for their great work. The organizers and volunteers of all the various PyCon and DjangoCon conferences are extraordinary. The Python community is the best one I’ve found in the tech industry.

Thank you.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

INTRODUCTION



"You've just done in two hours what it takes the three of us two days to do." My college roommate was working at a retail electronics store in the early 2000s. Occasionally, the store would receive a spreadsheet of thousands of product prices from other stores. A team of three employees would print the spreadsheet onto a thick stack of paper and split it among themselves. For each product price, they would look up their store's price and note all the products that their competitors sold for less. It usually took a couple of days.

"You know, I could write a program to do that if you have the original file for the printouts," my roommate told them when he saw them sitting on the floor with papers scattered and stacked all around.

After a couple of hours, he had a short program that read a competitor's price from a file, found the product in the store's database, and noted

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

whether the competitor was cheaper. He was still new to programming, so he spent most of his time looking up documentation in a programming book. The actual program took only a few seconds to run. My roommate and his co-workers took an extra-long lunch that day.

This is the power of computer programming. A computer is like a Swiss Army knife with tools for countless tasks. Many people spend hours clicking and typing to perform repetitive tasks, unaware that the machine they're using could do their job in seconds if they gave it the right instructions.

Who Is This Book For?

Software is at the core of so many of the tools we use today: nearly everyone uses social networks to communicate, virtually all people have internet-connected phones in their purse or pocket, and most office jobs involve interacting with a computer to get work done. As a result, the demand for people who can code has skyrocketed. Countless books, online tutorials, and developer boot camps promise to turn ambitious beginners into software engineers with six-figure salaries.

This book is not for those people. It's for everyone else.

On its own, this book won't turn you into a professional software developer any more than a few guitar lessons will turn you into a rock star. But if you're an office worker, administrator, academic, or anyone else who uses a computer for work or fun, you will learn the basics of programming so that you can automate simple tasks such as these:

- Moving and renaming thousands of files and sorting them into folders
- Filling out online forms—no typing required
- Downloading files or copying text from a website whenever it updates
- Having your computer text custom notifications to your phone
- Updating or formatting Excel spreadsheets
- Checking your email and sending out prewritten responses
- Creating databases and querying them for information
- Extracting text from images and audio files

These tasks are simple but time-consuming for humans, and they're often so trivial or specific that there's no ready-made software to perform them. Armed with a little bit of programming knowledge, however, you can have your computer do these tasks for you.

Coding Conventions Used in This Book

This book is not designed as a reference manual; it's a guide for beginners. The coding style sometimes goes against best practices (for example, some programs use global variables), but this trade-off makes the code simpler to learn. Sophisticated programming concepts—like object-oriented programming, list comprehensions, and generators—aren't covered because of the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

complexity they add. Veteran programmers may point out ways the code in this book could be changed to improve efficiency, but this book is mostly concerned with getting programs to work with the least amount of effort on your part.

What Is Programming?

Television shows and films often show programmers furiously typing cryptic streams of 1s and 0s on glowing screens, but modern programming isn't that mysterious. *Programming* is writing instructions for the computer to perform in a language the computer can understand. These instructions might crunch some numbers, modify text, look up information in files, or communicate with other computers over the internet.

All programs use basic instructions as building blocks. Here are a few of the most common ones, in English:

> "Do this; then do that."
>
> "If this condition is true, perform this action; otherwise, do that action."
>
> "Do this action exactly 27 times."
>
> "Keep doing that until this condition is true."

You can combine these building blocks to implement more intricate decisions too. For example, here are the programming instructions, called the *source code*, for a simple program written in the Python programming language. Starting at the top, the Python software runs lines of code (some of which are run only *if* a certain condition is true, or *else* Python runs some other line) until it reaches the bottom:

```
❶ password_file = open('SecretPasswordFile.txt')
❷ secret_password = password_file.read()
❸ print('Enter your password.')
  typed_password = input()
❹ if typed_password == secret_password:
    ❺ print('Access granted')
    ❻ if typed_password == '12345':
        ❼ print('That password is one that an idiot puts on their luggage.')
else:
  ❽ print('Access denied')
```

You might not know anything about programming, but you could probably make a reasonable guess at what the previous code does just by reading it. First, the file *SecretPasswordFile.txt* is opened ❶, and the secret password in it is read ❷. Then, the user is prompted to input a password (from the keyboard) ❸. These two passwords are compared ❹, and if they're the same, the program prints *Access granted* to the screen ❺. Next, the program checks whether the password is *12345* ❻ and hints that this choice might not be the best for a password ❼. If the passwords are not the same, the program prints *Access denied* to the screen ❽.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Programming is a creative task, as are painting, writing, knitting, and constructing LEGO castles. Like a blank canvas for painting, software has many constraints but endless possibilities. The difference between programming and other creative activities is that your computer comes with all the raw materials you need to program; you don't have to buy any additional canvas, paint, film, yarn, LEGO bricks, or electronic components. A decade-old laptop is more than powerful enough to write programs. Once you've written your program, you can copy it perfectly an infinite number of times. A knit sweater can be worn by only one person at a time, but a useful program can easily be shared online with the entire world.

What Is Python?

Python refers to both a programming language (with syntax rules for writing what is considered valid Python code) and the interpreter software that reads source code (written in the Python language) and performs its instructions. You can download Windows, macOS, and Linux versions of the Python interpreter for free at *https://python.org*.

There are several programming languages, each with their strengths and weaknesses. Debating which is best often leads to pointless arguments over matters of opinion. But in my opinion, Python is the best *first* language to learn if you are new to programming. Python has a gentle learning curve and readable syntax. It doesn't require learning dense concepts to do simple tasks. And if you want to go further into programming, learning a second language is easier when you first understand Python.

The name Python comes from the surreal British comedy group Monty Python, not from the snake. Python programmers are affectionately called Pythonistas, and both Monty Python and serpentine references usually pepper Python tutorials and documentation.

Common Myths About Programming

Programming has an intimidating reputation, and billion-dollar software companies are household names. Even the English word *code* has an association with secrecy and cryptic connotations. This leads many people to think that only a select few can program. But coding is a skill anyone can learn, and I'd like to address some of the more common myths directly.

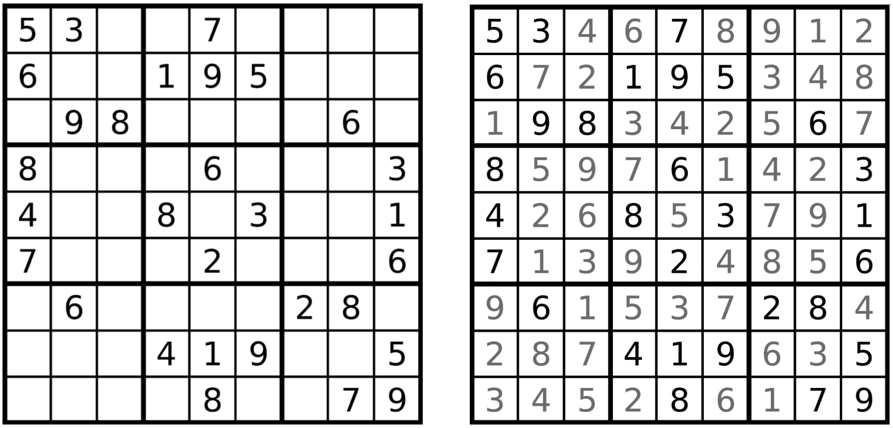
Programmers Don't Need to Know Much Math

The most common anxiety I hear about learning to program is the notion that it requires a lot of math. Actually, most programming doesn't require math beyond basic arithmetic. Programming requires deduction and paying attention to detail more than mathematics. In fact, being good at programming isn't that different from being good at solving Sudoku puzzles.

To solve a Sudoku puzzle, you must fill in the numbers 1 through 9 for each row, each column, and each 3 × 3 interior square of the full 9 × 9 board. The puzzle provides you with some numbers to start, and you can

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

find a solution by making deductions based on these numbers. In the puzzle shown in Figure 1, a 5 appears in the first and second rows, so it can't show up in these rows again. Therefore, in the upper-right grid, it must appear in the third row. Because the last column also already has a 5 in it, the 5 can't go to the right of the 6, so it must go to the left of the 6. Solving one row, column, or square will provide more clues for solving the rest of the puzzle, and as you fill in one group of numbers 1 to 9 and then another, you'll soon solve the entire grid.



5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

Figure 1: A new Sudoku puzzle (left) and its solution (right). Despite using numbers, Sudoku doesn't involve much math. (Images © Wikimedia Commons)

The fact that Sudoku involves numbers doesn't mean you have to be good at math to figure out the solution. The same is true of programming. Like solving a Sudoku puzzle, writing programs involves paying attention to details and breaking down a problem into individual steps. Similarly, when *debugging* programs (that is, finding and fixing errors), you'll patiently observe what the program is doing and find the cause of the bugs. And like all skills, the more you program, the better you'll become.

You Are Not Too Old to Learn Programming

The second most common anxiety I hear about learning to program is that people think they're too old to learn it. I read many internet comments from folks who think it's too late for them because they are already (gasp!) 23 years old. This is clearly not "too old" to learn to program; many people learn much later in life.

You don't need to have started as a child to become a capable programmer. But the image of programmers as whiz kids is a persistent one. Unfortunately, I contribute to this myth when I tell others that I was in grade school when I started programming.

However, programming is much easier to learn today than it was in the 1990s. Today, there are more books, better search engines, and many more online question-and-answer websites. On top of that, the programming languages themselves are far more user-friendly. For these reasons, *everything*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

I learned about programming in the years between grade school and high school graduation could be learned today in about a dozen weekends. My head start wasn't really much of a head start.

It's important to have a "growth mindset" about programming—in other words, understand that people develop programming skills through practice. They aren't just born as programmers, and being unskilled at programming now is not an indication that you can never become an expert.

AI Won't Replace Programmers

In the 1990s, nanotechnology promised to change everything. New manufacturing processes and scientific innovation would revolutionize society, the thinking went. Carbon nanotubes, buckyballs, and diamonds the price of pencil lead, all assembled one atom at a time out of plentiful carbon by germ-size nanobots, would pave the way for materials 10 times stronger than steel at a fraction of the weight and cost. This would lead to space elevators, medical miracles, and home appliances that could create anything, just like the replicators on *Star Trek*! It would mean an end to economic scarcity, world hunger, and war. It would bring on a new age of enlightenment—as long as the nanobots didn't turn on their human creators in a tiny robot uprising. And the technology was only 10 years away!

Of course, the hype never happened. Real innovations certainly occurred at the nanoscale (the smartphone in your pocket uses a number of them). But the Star Trek replicators and other grandiose promises didn't arrive, and the excitement over nanotechnology deflated to more realistic proportions.

Let's talk about AI.

Personal computers changed everything. The internet changed everything. Social media changed everything. Smartphones changed everything. Cryptocurrency did not change everything, but it did reveal which of your cousins and co-workers were susceptible to get-rich-quick scams. Today, AI is the latest marvel to emerge from the tech industry. People use the term *AI* to mean everything from chess-playing computers to chatbots, so-called "expert systems," and machine learning. In this book, I'll use the term *large language model (LLM)*, which is the conceptual category behind OpenAI's ChatGPT, Gemini, Facebook's LLaMa, and other generative text systems.

LLMs have caused many breathless claims and questions. Is AI going to take all of our jobs? Is it still worth learning to code? Are the AIs alive, and can I survive the AI-robot uprising?

LLM technology is exciting, but to make it useful, we need to set realistic expectations so that we can avoid falling prey to sensationalist journalism and questionable "investment opportunities." I hope I can deflate the hype and give you a more realistic view of how LLMs can help you learn to code. Let's get some of these misconceptions out of the way right now:

- LLMs are not conscious or sentient.
- LLMs will not replace human software engineers.
- LLMs do not alleviate the need to learn programming.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

- LLMs will not replace most human jobs (though this won't prevent your manager from thinking they can and laying you off anyway).
- LLMs are far from perfect, and are even *often* wrong.

Those who insist on these misconceptions get their information from science fiction movies and internet videos, not from experience with actual LLMs. I highly recommend Simon Willison, Python Software Foundation board member and co-creator of the Django Web Framework, for his writing about AI at *https://simonwillison.net/about*. You can watch his sober and illuminating PyCon 2024 keynote speech on LLMs at *https://autbor.com/pycon2024keynote*.

How could LLMs help you as a programmer? What is obvious at this early stage is that learning to communicate with LLMs (so-called *prompt engineering*) is a skill, just like learning to effectively use a search engine is a skill. LLMs are not people, and you need to learn how to phrase your questions to get relevant and reliable answers. When LLMs confidently make up incorrect answers, we say that they are *hallucinating*. But this is just another way of anthropomorphizing an algorithm. LLMs don't think; they generate text. That is to say, LLMs are *always* hallucinating, even when their answers happen to be correct.

LLMs make large, obvious mistakes and simple, subtle mistakes, however. If you use them as a learning aid, you must vigilantly check everything the LLM tells you, big or small. It's entirely valid to choose to forego learning with LLMs altogether. At this point, the effectiveness of using LLMs in education is unproven. We don't know for sure in what situations LLMs are useful as learning tools, or even if their benefits outweigh their costs.

It's no wonder that so many buy into the hype of LLMs. We carry devices in our pockets that scientists of the last century would consider supercomputers. They connect us to a global network of information (and misinformation). They can identify their position anywhere on Earth by listening to satellites in outer space. Software can already do seemingly magical things, but software isn't magic. And by learning to program, you'll get a far more grounded idea of what computers are capable of versus what is just hype.

About This Book

The first part of this book teaches you how to program in Python. The second part covers various software libraries for automating different kinds of tasks. I recommend reading the chapters of Part I in order, then skipping to the chapters in Part II that interest you. Here's a brief rundown of what you'll find in each chapter.

Part I: Programming Fundamentals

Chapter 1: Python Basics Covers expressions, the most basic type of Python instruction, and how to use the Python interactive shell software to experiment with code.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Chapter 2: if-else and Flow Control Explains how to make programs decide which instructions to execute so that your code can intelligently respond to different conditions.

Chapter 3: Loops Explains how to make programs repeat instructions a set number of times, or for as long as a certain condition holds.

Chapter 4: Functions Instructs you on how to define your own functions so that you can organize your code into more manageable chunks.

Chapter 5: Debugging Shows how to use Python's various bug-finding and bug-fixing tools.

Chapter 6: Lists Introduces the list data type and explains how to organize data.

Chapter 7: Dictionaries and Structuring Data Introduces the dictionary data type and shows you more powerful ways to organize data.

Chapter 8: Strings and Text Editing Covers working with text data (called *strings* in Python).

Part II: Automating Tasks

Chapter 9: Text Pattern Matching with Regular Expressions Covers how Python can manipulate strings and search for text patterns with regular expressions.

Chapter 10: Reading and Writing Files Explains how your program can read the contents of text files and save information to files on your hard drive.

Chapter 11: Organizing Files Shows how Python can copy, move, rename, and delete large numbers of files much faster than a human user can. Also explains compressing and decompressing files.

Chapter 12: Designing and Deploying Command Line Programs Explains how you can package your Python programs to easily run them either on your own computer or on co-workers' computers.

Chapter 13: Web Scraping Shows how to write programs that can automatically download web pages and parse them for information. This is called *web scraping*.

Chapter 14: Excel Spreadsheets Covers programmatically manipulating Excel spreadsheets so that you don't have to read them. This is helpful when the number of documents you have to analyze is in the hundreds or thousands.

Chapter 15: Google Sheets Covers how to read and update Google Sheets, a popular web-based spreadsheet application, using Python.

Chapter 16: SQLite Databases Explains how to use relational databases with SQLite, the tiny but powerful open source database that comes with Python.

Chapter 17: PDF and Word Documents Covers programmatically reading Word and PDF documents.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Chapter 18: CSV, JSON, and XML Files Continues to explain how to programmatically manipulate documents, now discussing the data serialization formats CSV, JSON, XML, YAML, and TOML.

Chapter 19: Keeping Time, Scheduling Tasks, and Launching Programs Explains how Python programs handle time and dates and how to schedule your computer to perform tasks at certain times. Also shows how your Python programs can launch non-Python programs.

Chapter 20: Sending Email, Texts, and Push Notifications Explains how to write programs that can notify you via email or mobile communications, or send these messages to others.

Chapter 21: Making Graphs and Manipulating Images Explains how to programmatically manipulate images, such as JPEG or PNG files, and work with the Matplotlib graph-making library.

Chapter 22: Recognizing Text in Images Covers how to extract text from images and scanned documents for further processing with the PyTesseract package.

Chapter 23: Controlling the Keyboard and Mouse Explains how to programmatically control the mouse and keyboard to automate clicks and keypresses.

Chapter 24: Text-to-Speech and Speech Recognition Engines Covers how to use advanced computer science packages to not only generate spoken audio from text, but also convert spoken audio to text.

You can download source code and other resources for the examples in this book at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*. To see many of this book's programs in action, visit *https://autbor.com/3*.

Downloading and Installing Python

You can download Python for Windows, macOS, and Ubuntu Linux for free at *https://python.org/downloads*.

The download page detects your operating system and recommends the download package for your computer. There are unofficial Python installers for Android and iOS mobile operating systems, but those are beyond the scope of this book. Windows, macOS, and Linux have their own installation options as well. Download the installer for your operating system and run the program to install the Python interpreter software.

New versions of Python or your operating system may change the steps needed to install Python. If you encounter difficulties, you can consult *https://autbor.com/install/* for up-to-date instructions.

Downloading and Installing Mu

While the *Python interpreter* is the software that runs your Python programs, the *Mu editor software* is where you'll enter your programs, much the way you enter text in a word processor. You can download Mu from *https://codewith.mu*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

On Windows and macOS, download the installer for your operating system, then run it by double-clicking the installer file. If you are on macOS, running the installer opens a window where you must drag the Mu icon to the Applications folder icon to continue the installation. If you are on Ubuntu, you'll need to install Mu as a Python package. In that case, click the Instructions button in the Python Package section of the download page.

Starting Mu

Once it's installed, you can start Mu:

- On Windows, click the Start icon in the lower-left corner of your screen, enter **Mu Editor** in the search box, and select it.
- On macOS, open the Finder window, click **Applications**, and then click **mu-editor**. You can also run **Mu Editor** from Spotlight.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter `python3 -m mu`.

The first time Mu runs, a Select Mode window will appear with options for Adafruit CircuitPython, BBC micro:bit, Pygame Zero, Python 3, and others. Select **Python 3**. You can always change the mode later by clicking the **Mode** button at the top of the editor window.

Starting IDLE

This book uses Mu as an editor and interactive shell. However, you can use any number of editors for writing Python code. The *interactive development environment (IDLE)* software installs along with Python, and it can serve as a second editor if for some reason you can't get Mu installed or working. Let's start IDLE now (assuming Python 3.13 is the version you installed):

- On Windows, click the Start icon in the lower-left corner of your screen, enter **IDLE** in the search box, and select **IDLE (Python 3.13 64-bit)**.
- On macOS, open the Finder window, click **Applications**, click **Python 3.13**, and then click the IDLE icon. You can also run IDLE from Spotlight.
- On Ubuntu, select **Applications ▸ Accessories ▸ Terminal** and then enter `idle`. (You may also be able to click **Show Apps** at the bottom of the Ubuntu sidebar and then click **IDLE**.)

The Interactive Shell

When you run Mu, the window that appears is called the *file editor* window. You can open the *interactive shell* by clicking the REPL button. A shell is a program that lets you enter instructions into the computer, much like the Terminal or Command Prompt on macOS and Windows, respectively.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Python's interpreter software will immediately run any instructions you enter into the Python interactive shell.

In Mu, the interactive shell is a pane in the lower half of the window with something like the following text:

```
Jupyter QtConsole
Python 3
Type 'copyright', 'credits' or 'license' for more information
IPython -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

If you run IDLE, the interactive shell is the window that first appears. It should be mostly blank except for text that looks something like this:

```
>>>
```

`In [1]:` and >>> are called *prompts*. The examples in this book will use the >>> prompt to represent the interactive shell, since it's more common. If you run Python from the Terminal or Command Prompt, they'll use the >>> prompt as well. The `In [1]:` prompt was invented by Jupyter Notebook, another popular Python editor.

For example, enter the following into the interactive shell next to the prompt:

```
>>> print('Hello, world!')
```

After you write the line and press ENTER, the interactive shell should display this in response:

```
>>> print('Hello, world!')
Hello, world!
```

You've just given the computer an instruction, and it did what you told it to do!

How to Find Help

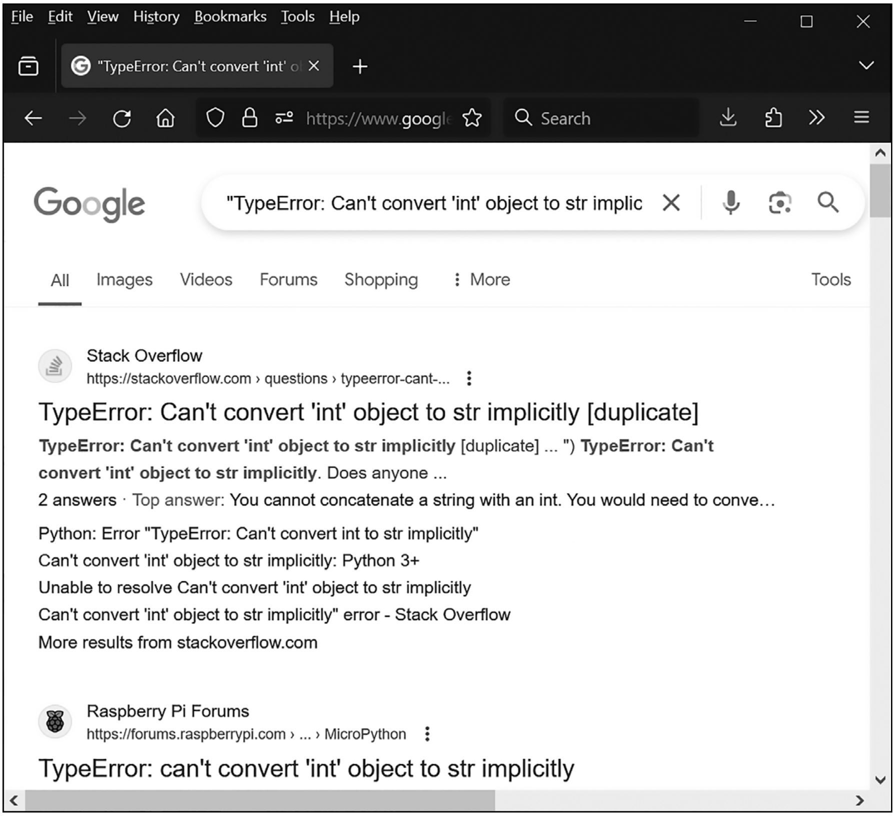
Programmers tend to learn by searching the internet for answers to their questions. This is quite different from the way many people are accustomed to learning—through an in-person teacher who lectures and can answer questions. What's great about using the internet as a schoolroom is that there are whole communities of folks who can help you solve your problems. Indeed, your questions have probably already been answered, and the answers are waiting online for you to find them. If you encounter an error message or have trouble making your code work, you won't be the first person to have your problem, and finding a solution is easier than you might think.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

For example, let's cause an error on purpose: enter **'42' + 3** into the interactive shell. You don't need to know what this instruction means right now, but the result should look like this:

```
  >>> '42' + 3
❶ Traceback (most recent call last):
    File "<pyshell#0>", line 1, in <module>
      '42' + 3
❷ TypeError: Can't convert 'int' object to str implicitly
  >>>
```

The error message ❷ appears because Python couldn't understand your instruction. The traceback part ❶ of the error message shows the specific instruction and line number that Python had trouble with. If you're not sure what to make of a particular error message, search for it online. Enter **"TypeError: Can't convert 'int' object to str implicitly"** (including the quotation marks) into your favorite search engine, and you should see tons of links explaining what the error message means and what causes it, as shown in Figure 2.



"TypeError: Can't convert 'int' object to str implic

Stack Overflow
https://stackoverflow.com › questions › typeerror-cant-...

TypeError: Can't convert 'int' object to str implicitly [duplicate]

TypeError: Can't convert 'int' object to str implicitly [duplicate] ... ") **TypeError: Can't convert 'int' object to str implicitly**. Does anyone ...

2 answers · Top answer: You cannot concatenate a string with an int. You would need to conve...

Python: Error "TypeError: Can't convert int to str implicitly"
Can't convert 'int' object to str implicitly: Python 3+
Unable to resolve Can't convert 'int' object to str implicitly
Can't convert 'int' object to str implicitly" error - Stack Overflow
More results from stackoverflow.com

Raspberry Pi Forums
https://forums.raspberrypi.com › ... › MicroPython

TypeError: can't convert 'int' object to str implicitly

Figure 2: The Google results for an error message can be very helpful.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You'll often find that someone else had the same question as you and that some other helpful person has already answered it. No one person can know everything about programming, so an everyday part of any software developer's job is looking up answers to technical questions.

Asking Smart Programming Questions

If you can't find the answer by searching online, try asking people in a web forum such as Stack Overflow (*https://stackoverflow.com*) or the "learn programming" subreddit at *https://reddit.com/r/learnprogramming*. But keep in mind there are smart ways to ask programming questions that help others help you. To begin with, be sure to read the Frequently Asked Questions sections at these websites about the proper way to post questions.

When asking programming questions, remember to do the following:

- Explain what you are trying to do, not just what you did. This lets your helper know if you are on the wrong track.
- Specify the point at which the error happens. Does it occur at the very start of the program or only after you do a certain action?
- Copy and paste the *entire* error message and your code to *https://pastebin.com* or *https://gist.github.com*. These websites make it easy to share large amounts of code with people online, without losing any text formatting. You can then put the URL of the posted code in your email or forum post. For example, here are the locations of some pieces of code I've posted: *https://pastebin.com/2k3LqDsd* and *https://gist.github.com/asweigart/6912168*.
- Explain what you've already tried to do to solve your problem. This tells people you've already put in some work to figure things out on your own.
- List the version of Python you're using. Also, say which operating system and version you're running.
- If the error came up after you made a change to your code, explain exactly what you changed.
- Say whether you're able to reproduce the error every time you run the program or whether it happens only after you perform certain actions. In the latter case, also explain what those actions are.

Always follow good online etiquette as well. For example, don't post your questions in all caps or make unreasonable demands of the people trying to help you.

You can find more information on how to ask for programming help in the blog post at *https://autbor.com/help*. I love helping people discover Python. I write programming tutorials on my blog at *https://inventwithpython.com/blog*, and you can contact me with questions at *al@inventwithpython.com*, although you may get a faster response by posting your questions to *https://reddit.com/r/inventwithpython*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Summary

For most people, their computer is just an appliance instead of a tool. But by learning how to program, you'll gain access to one of the most powerful tools of the modern world, and you'll have fun along the way. Programming isn't brain surgery—it's fine for amateurs to experiment and make mistakes.

This book assumes you have zero programming knowledge and will teach you quite a bit, but you may have questions beyond its scope. Remember that asking effective questions and knowing how to find answers are invaluable tools on your programming journey.

Let's begin!

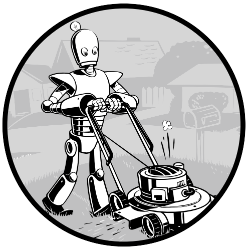
Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

PART I

PROGRAMMING FUNDAMENTALS

1

PYTHON BASICS



The Python programming language has a wide range of syntactical constructions, standard library functions, and interactive development environment features. Fortunately, you can ignore most of that; you just need to learn enough to write some handy little programs.

To accomplish this, however, you'll have to master some programming concepts. Like a wizard in training, you might think these concepts seem tedious, but with some practice, they'll enable you to command your computer like a magic wand and perform incredible feats.

This chapter has a few examples that encourage you to enter code into the *interactive shell*, also called the *read-evaluate-print-loop (REPL)*, which lets you run (or *execute*) Python instructions one at a time and instantly shows you the results. Using the interactive shell is great for learning what basic Python instructions do, so give it a try as you follow along. You'll remember the things you do much better than the things you only read.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Entering Expressions into the Interactive Shell

You can run the interactive shell by launching the Mu editor. This book's Introduction provides setup instructions for downloading and installing it. On Windows, open the Start menu, enter **Mu**, and open the Mu app. On macOS, open your Applications folder and double-click **Mu**. Click the **New** button and save an empty file as *blank.py*. When you run this blank file by clicking the **Run** button or pressing F5, it will open the interactive shell, which will open as a new pane at the bottom of the Mu editor's window. You should see a `>>>` prompt in the interactive shell.

You can also run the interactive shell from the command line Terminal (on macOS and Linux) or Windows Terminal (on Windows, where you could also use the older Command Prompt application). After opening these command line windows, enter **`python`** (on Windows) or **`python3`** (on macOS and Linux). You'll see the same `>>>` prompt for the interactive shell. If you want to run a program, run `python` or `python3` followed by the name of the program's *.py* file, such as `python blank.py`. Be sure you don't run `python` on macOS's Terminal, as this may launch the older, backward-incompatible Python 2.7 version on certain versions of macOS. You may even see a message saying `WARNING: Python 2.7 is not recommended`. Exit the 2.7 interactive shell and run `python3` instead.

Enter **`2 + 2`** at the prompt to have Python do some simple math. The Mu window should now look like this:

```
>>> 2 + 2
4
>>>
```

In Python, `2 + 2` is called an *expression*, which is the most basic kind of programming instruction in the language. Expressions consist of *values* (such as `2`) and *operators* (such as `+`), and they can always *evaluate* (that is, reduce) down to a single value. That means you can use expressions anywhere in Python code that you could also use a value.

In the previous example, `2 + 2` is evaluated down to a single value, `4`. A single value with no operators is also considered an expression, though it evaluates only to itself, as shown here:

```
>>> 2
2
```

The Mu editor has a REPL button that shows an interactive shell with a prompt that looks like `In [1]:`. The popular Jupyter Notebook editor uses this kind of interactive shell. You can use this interactive shell the same way as the normal Python interactive shell with the `>>>` prompt. REPLs are not unique to Python; many programming languages also offer REPLs so that you can experiment with their code.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

ERRORS ARE OKAY!

The best thing about computers is that they carry out the exact instructions you give them. This is also the worst thing about computers. Computers can't use common sense to figure out what you intended to do. Programs will crash if they contain code the computer can't understand, which will cause Python to show an error message. Error messages don't damage your computer, though, so don't be afraid to make mistakes. A *crash* just means the program unexpectedly stopped running.

Get used to seeing error messages, because you'll constantly encounter them (even if you have decades of programming experience). Error messages are often vague and not meant to be immediately understood by beginners. If you want to know more about an error, you can search for the exact error message text online for more information. Note that if you are using the Mu editor, the keyboard shortcut for copying highlighted text in the interactive shell pane to the clipboard is CTRL-SHIFT-C, while the shortcut for copying text in the main file editor pane is the standard CTRL-C. You can also check out the resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* to see a list of common Python error messages and their meanings.

Programming involves some math operations you might not be familiar with:

- Exponentiation (or *to the power of*) is multiplying a number by itself repeatedly, just like multiplication is adding a number to itself repeatedly. For example, *two to the power of four* (or *two to the fourth power*), written as or 2^4 or `2 ** 4`, is the number two multiplied by itself four times: $24 = 2 \times 2 \times 2 \times 2 = 16$.
- Modular arithmetic is similar to the remainder result of division. For example, `14 % 4` evaluates to `2` because 14 divided by 4 is 3 with remainder 2. Even though Python's modulo operator is `%`, modular arithmetic has nothing to do with percentages.
- Integer division is the same as regular division except the result is rounded down. For example, `25 / 8` is `3.125` but `25 // 8` is `3`, and `29 / 10` is `2.9` but `29 // 10` is `2`.

You can use plenty of other operators in Python expressions too. For example, Table 1-1 lists all the math operators in Python.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Table 1-1: Math Operators from Highest to Lowest Precedence

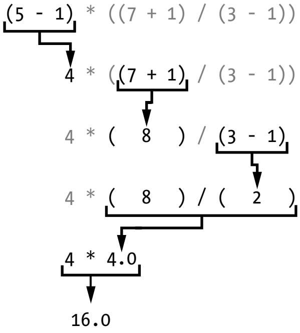
Operator	Operation	Example	Evaluates to . . .
`**`	Exponentiation	`2 ** 3`	`8`
`%`	Modulus/remainder	`22 % 8`	`6`
`//`	Integer division	`22 // 8`	`2`
`/`	Division	`22 / 8`	`2.75`
`*`	Multiplication	`3 * 5`	`15`
`-`	Subtraction	`5 - 2`	`3`
`+`	Addition	`2 + 2`	`4`

The *order of operations* (also called *precedence*) of Python math operators is similar to that of mathematics. The ** operator is evaluated first; the *, /, //, and % operators are evaluated next, from left to right; and the + and - operators are evaluated last (also from left to right). You can use parentheses to override the usual precedence if you need to. Whitespace in between the operators and values doesn't matter in Python, except for the indentation at the beginning of the line. But the *convention*, or unofficial rule, is to have a single space in between operators and values. Enter the following expressions into the interactive shell:

```
>>> 2 + 3 * 6
20
>>> (2 + 3) * 6
30
>>> 48565878 * 578453
28093077826734
>>> 2 ** 8
256
>>> 23 / 7
3.2857142857142856
>>> 23 // 7
3
>>> 23 % 7
2
>>> 2      +           2
4
>>> (5 - 1) * ((7 + 1) / (3 - 1))
16.0
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

In each case, you as the programmer must enter the expression, but Python does the hard part of evaluating it. Python will keep evaluating parts of the expression until it becomes a single value:



These rules for putting operators and values together to form expressions are a fundamental part of Python as a programming language, just like the grammar rules that help us communicate. Here's an example:

This is a grammatically correct English sentence.

This grammatically is sentence not English correct a.

The second line is difficult to parse because it doesn't follow the rules of English. Similarly, if you enter a bad Python instruction, Python won't be able to understand it and will display a SyntaxError error message, as shown here:

```
>>> 5 +
  File "<stdin>", line 1
    5 +
      ^
SyntaxError: invalid syntax
>>> 42 + 5 + * 2
  File "<stdin>", line 1
    42 + 5 + * 2
             ^
SyntaxError: invalid syntax
```

You can always test whether an instruction works by entering it into the interactive shell. Don't worry about breaking the computer; the worst that could happen is that Python responds with an error message. Professional software developers get error messages all the time while writing code.

The Integer, Floating-Point, and String Data Types

Remember that expressions are just values combined with operators, and they always evaluate to a single value. A *data type* is a category for values, and every value belongs to exactly one data type. The most common data types in Python are listed in Table 1-2. The values `-2` and `30`, for example, are said to be *integer* values. The integer (or *int*) data type indicates values that are whole numbers. Numbers with a decimal point, such as `3.14`, are called *floating-point numbers* (or *floats*). Note that even though the value `42` is an integer, the value `42.0` would be a floating-point number. Programmers often use *number* to refer to ints and floats collectively, although *number* itself is not a Python data type.

One subtle detail about Python is that any math performed using an int and a float results in a float, not an int. While `3 + 4` evaluates to the integer `7`, the expression `3 + 4.0` evaluates to the floating-point number `7.0`. Any division between two integers with the `/` division operator results in a float as well. For example, `16 / 4` evaluates to `4.0` and not `4`. Most of the time, this information doesn't matter for your program, but knowing it will explain why your numbers may suddenly gain a decimal point.

Table 1-2: Common Data Types

Data type	Examples
Integer (int)	`-2, -1, 0, 1, 2, 3, 4, 5`
Floating-point number (float)	`-1.25, -1.0, -0.5, 0.0, 0.5, 1.0, 1.25`
String (str)	`'a', 'aa', 'aaa', 'Hello!', '11 cats', '5'`

Python programs can also have text values called *strings*, or *strs* (pronounced *stirs*). Always surround your string in single-quote (`'`) characters (as in `'Hello'` or `'Goodbye cruel world!'`) so that Python knows where the string begins and ends. You can even have a string with no characters in it, `''`, called a *blank string* or an *empty string*. Strings are explained in greater detail in Chapter 8.

You may see the error message `SyntaxError: unterminated string literal`, as in this example:

```
>>> 'Hello, world!
SyntaxError: unterminated string literal (detected at line 1)
```

This error means you probably forgot the final single-quote character at the end of the string.

String Concatenation and Replication

The meaning of an operator may change based on the data types of the values next to it. For example, `+` is the addition operator when it operates on two integers or floating-point values. However, when `+` is used to combine two string values, it joins the strings as the *string concatenation* operator. Enter the following into the interactive shell:

```
>>> 'Alice' + 'Bob'
'AliceBob'
```

The expression evaluates down to a single, new string value that combines the text of the two strings. However, if you try to use the + operator on a string and an integer value, Python won't know how to handle this and will display an error message:

```
>>> 'Alice' + 42
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    'Alice' + 42
TypeError: can only concatenate str (not "int") to str
```

The error message `can only concatenate str (not "int") to str` means that Python thought you were trying to concatenate an integer to the string `'Alice'`. Your code will have to explicitly convert the integer to a string because Python cannot do this automatically. (I'll explain how to convert between data types in "Dissecting the Program" on page XX, where we talk about the `str()`, `int()`, and `float()` functions.)

The * operator multiplies two integer or floating-point values. But when the * operator is used on one string value and one integer value, it becomes the *string replication* operator. Enter a string multiplied by a number into the interactive shell to see this in action:

```
>>> 'Alice' * 5
'AliceAliceAliceAliceAlice'
```

The expression evaluates down to a single string value that repeats the original string a number of times equal to the integer value. String replication is a useful trick, but it's not used as often as string concatenation.

The * operator can only be used with two numeric values (for multiplication), or one string value and one integer value (for string replication). Otherwise, Python will just display an error message, such as the following:

```
>>> 'Alice' * 'Bob'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    'Alice' * 'Bob'
TypeError: can't multiply sequence by non-int of type 'str'
>>> 'Alice' * 5.0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    'Alice' * 5.0
TypeError: can't multiply sequence by non-int of type 'float'
```

It makes sense that Python wouldn't understand these expressions: you can't multiply two words, and it's hard to replicate an arbitrary string a fractional number of times.

Expressions, data types, and operators may seem abstract to you right now, but as you learn more about these concepts, you'll be able to create increasingly sophisticated programs that do math on data pulled from spreadsheets, websites, the output of other programs, and other places.

Storing Values in Variables

A *variable* is like a box in the computer's memory where you can store a single value. If you want to use the result of an evaluated expression later in your program, you can save it inside a variable.

Assignment Statements

You'll store values in variables with an *assignment statement*. An assignment statement consists of a variable name, an equal sign (called the *assignment operator*), and the value to be stored. If you enter the assignment statement `spam = 42`, a variable named `spam` will have the integer value `42` stored in it.

You can think of a variable as a labeled box that a value is placed in, but Chapter 6 explains how a name tag attached to the value might be a better metaphor. Both are shown in Figure 1-1.

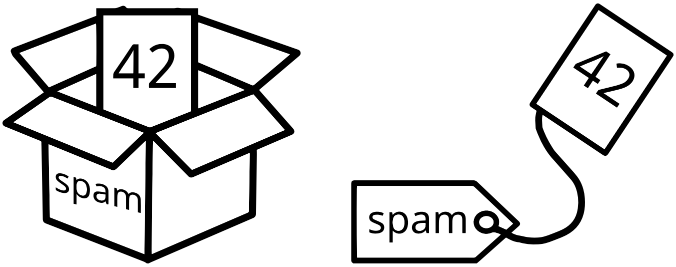


Figure 1-1: The code `spam = 42` is like telling the program, "The variable `spam` now has the integer value `42` in it."

For example, enter the following into the interactive shell:

```
❶ >>> spam = 40
  >>> spam
  40
  >>> eggs = 2
❷ >>> spam + eggs
  42
  >>> spam + eggs + spam
  82
❸ >>> spam = spam + 2
  >>> spam
  42
```

A variable is *initialized* (or created) the first time a value is stored in it ❶. After that, you can use it in expressions with other variables and values ❷. When a variable is assigned a new value ❸, the old value is forgotten, which is why `spam` evaluated to `42` instead of `40` at the end of the example. This is called *overwriting* the variable. Enter the following code into the interactive shell to try overwriting a string:

```
>>> spam = 'Hello'
>>> spam
```

```
'Hello'
>>> spam = 'Goodbye'
>>> spam
'Goodbye'
```

Just like the box in Figure 1-1, the spam variable in Figure 1-2 stores 'Hello' until you replace the string with 'Goodbye'.

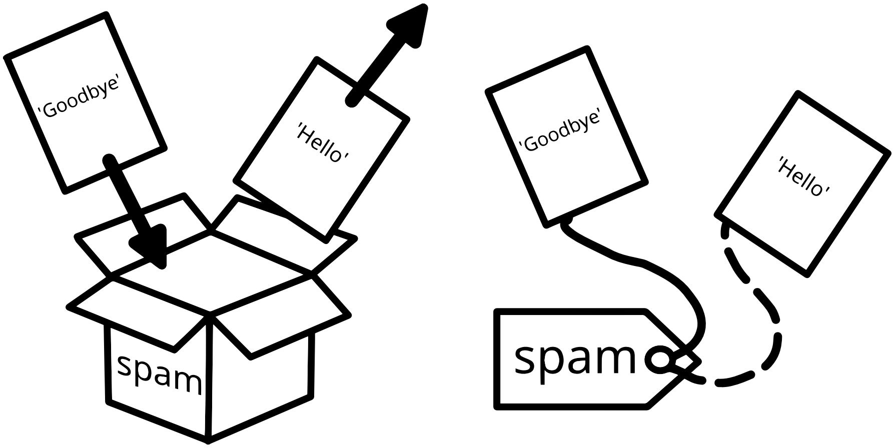


Figure 1-2: When a new value is assigned to a variable, the old one is forgotten.

You can also think of overwriting a variable as reassigning the name tag to a new value.

Variable Names

A good variable name describes the data it contains. Imagine that you moved to a new house and labeled all of your moving boxes as *Stuff*. You'd never find anything! Most of this book's examples (and Python's documentation) use generic variable names like spam, eggs, and bacon, which come from the Monty Python "Spam" sketch. But in your programs, descriptive names will help make your code more readable.

Though you can name your variables almost anything, Python does have some naming restrictions. Your variable name must obey the following four rules:

- It can't have spaces.
- It can use only letters, numbers, and the underscore (_) character.
- It can't begin with a number.
- It can't be a Python keyword, such as if, for, return, or other keywords you'll learn in this book.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Table 1-3 shows examples of legal variable names.

Table 1-3: Valid and Invalid Variable Names

Valid variable names	Invalid variable names
`current_balance`	`current-balance` (hyphens are not allowed)
`currentBalance`	`current balance` (spaces are not allowed)
`account4`	`4account` (can't begin with a number)
`_42`	`42` (can begin with an underscore but not a number)
`TOTAL_SUM`	`TOTAL_$UM` (special characters like $ are not allowed)
`hello`	`'hello'` (special characters like ' are not allowed)

Variable names are case-sensitive, meaning that `spam`, `SPAM`, `Spam`, and `sPaM` are four different variables. It is a Python convention to start your variables with a lowercase letter: `spam` instead of `Spam`.

CODE STYLE OPINIONS AND PEP 8

Previous editions of this book used *camelCase* instead of underscores to separate words in variable names; that is, variables `lookedLikeThis` instead of `looking_like_this`. The latter form is called *snake_case* because the underscores between words look like little snakes (while the uppercase letters in camelCase look like the humps on a camel). Some experienced programmers may point out that the official Python code style document, PEP 8, says that underscores should be used. I unapologetically prefer camelCase and point to the "A Foolish Consistency Is the Hobgoblin of Little Minds" section in PEP 8 itself as my defense:

> *Consistency with the style guide is important. But most importantly: know when to be inconsistent—sometimes the style guide just doesn't apply. When in doubt, use your best judgment.*

The computer doesn't care which style you use, and PEP 8 is not a stone tablet of irrefutable commandments. It doesn't matter which style you use as long as you use the same style consistently. To prove this, I've rewritten the code in this book to use snake_case because it truly doesn't matter either way.

Your First Program

While the interactive shell is good for running Python instructions one at a time, to write entire Python programs you'll enter the instructions into the file editor. The *file editor* is similar to text editors such as Notepad and TextMate, but it has some features specifically for entering source code. To open a new file in Mu, click the **New** button on the top row.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The tab that appears should contain a cursor awaiting your input, but it's different from the interactive shell, which runs Python instructions as soon as you press ENTER. The file editor lets you enter many instructions, save the file, and run the program. Here's how you can tell the difference between the two:

- The interactive shell will always be the one with the `>>>` or `In [1]:` prompt.
- The file editor won't have the `>>>` or `In [1]:` prompt.

Now it's time to create your first program! When the file editor window opens, enter the following into it:

```
# This program says hello and asks for my name.

print('Hello, world!')
print('What is your name?')  # Ask for their name.
my_name = input('>')
print('It is good to meet you, ' + my_name)
print('The length of your name is:')
print(len(my_name))
print('What is your age?')  # Ask for their age.
my_age = input('>')
print('You will be ' + str(int(my_age) + 1) + ' in a year.')
```

Once you've entered your source code, save it so that you won't have to retype it each time you start Mu. Click **Save**, enter **hello.py** in the File Name field, and then click **Save**.

You should save your programs every once in a while as you type them. That way, if the computer crashes or you accidentally exit Mu, you won't lose the code. As a shortcut, you can press CTRL-S on Windows and Linux or ⌘-S on macOS to save your file.

Once you've saved, let's run our program. Press the F5 key or click the **Run** button. Enter your name when your program asks for it. The program's output in the interactive shell should look something like this:

```
Hello, world!
What is your name?
>Al
It is good to meet you, Al
The length of your name is:
2
What is your age?
>4
You will be 5 in a year.
>>>
```

When there are no more lines of code to execute, the Python program *terminates*; that is, it stops running. (You can also say that the Python program *exits*.) The Mu editor displays the `>>>` interactive shell prompt after the program terminated, in case you'd like to enter some further Python code.

You can close the file editor by clicking the **X** on the file's tab, just like closing a browser tab. To reload a saved program, click **Load** from the menu. Do that now, and in the window that appears, choose **hello.py** and click the **Open** button. Your previously saved *hello.py* program should open in the file editor window.

You can view the step-by-step execution of a program using the Python Tutor visualization tool at *http://pythontutor.com*. Click the forward button to move through each step of the program's execution. You'll be able to see how the variables' values and the output change.

Dissecting the Program

With your new program open in the file editor, let's take a quick tour of the Python instructions it uses by looking at what each line of code does.

Comments

The following line is called a *comment*:

```
# This program says hello and asks for my name.
```

Python ignores comments, and you can use them to write notes or remind yourself what the code is trying to do. Any text for the rest of the line following a hash mark (`#`) is part of a comment.

Sometimes programmers will put a `#` in front of a line of code to temporarily remove it while testing a program. This is called *commenting out* code, and it can be useful when you're trying to figure out why a program isn't working. You can remove the `#` later when you are ready to put the line back in.

Python also ignores the blank line after the comment. You can add as many blank lines to your program as you want. This spacing can make your code easier to read, like paragraphs in a book.

The print() Function

The `print()` function displays the string value inside its parentheses on the screen:

```
print('Hello, world!')
print('What is your name?')  # Ask for their name.
```

The line `print('Hello, world!')` means "Print out the text in the string `'Hello, world!'`." When Python executes this line, you say that Python is *calling* the `print()` function and the string value is being *passed* to the function. A value that is passed to a function call is an *argument*. Notice that the quotes are not printed to the screen. They just mark where the string begins and ends; they are not part of the string value's text.

NOTE *You can also use this function to display a blank line on the screen; call* `print()` *with nothing in between the parentheses.*

When you write a function name, the opening and closing parentheses at the end identify it as the name of a function. This is why in this book, you'll see `print()` rather than `print`. It's a standard convention to have no spaces in between the function name and the opening parentheses, even though Python doesn't require this. Chapter 3 describes functions in more detail.

The input() Function

The `input()` function waits for the user to type some text on the keyboard and press ENTER:

```
my_name = input('>')
```

This function call evaluates to a string identical to the user's text, and the rest of the code assigns the `my_name` variable to this string value. The `'>'` string passed to the function causes the > prompt to appear, which serves as an indicator to the user that they are expected to enter something. Your programs don't have to pass a string to the `input()` function; if you call `input()`, the program will wait for the user's text without displaying any prompt.

THE > AND >>> PROMPTS

You can pass any string to `input()` to change the prompt that appears when your program runs. Calling `input('>')` puts an angle bracket > on the screen as the prompt. This is different from the >>> prompt that appears in the Python interactive shell. In this book, the >>> prompt indicates a response to the Python interactive shell and the > prompt indicates a response to a program running the `input('>')` call. My choice of `'>'` was arbitrary; you can use any prompt you want or no prompt at all.

You can think of the `input()` function call as an expression that evaluates to whatever string the user typed. If the user entered `'Al'`, the assignment statement would effectively be `my_name = 'Al'`.

If you call `input()` and see an error message, like `NameError: name 'Al' is not defined`, the problem is that you're running the code with Python 2 instead of Python 3.

The Greeting Message

The following call to `print()` contains the expression `'It is good to meet you, ' + my_name` between the parentheses:

```
print('It is good to meet you, ' + my_name)
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Remember that expressions can always evaluate to a single value. If `'Al'` is the value stored in `my_name`, then this expression evaluates to `'It is good to meet you, Al'`. This single string value is then passed to `print()`, which prints it on the screen.

The len() Function

You can pass the `len()` function a string value (or a variable containing a string), and the function evaluates to the integer value of the number of characters in that string:

```
print('The length of your name is:')
print(len(my_name))
```

Enter the following into the interactive shell to try this:

```
>>> len('hello')
5
>>> len('My very energetic monster just scarfed nachos.')
46
>>> len('')
0
```

Just like in those examples, `len(my_name)` evaluates to an integer. We say that the `len()` function call *returns* or *outputs* this integer value, and the value is the function call's *return value*. It is then passed to `print()` to be displayed on the screen. The `print()` function allows you to pass it either integer values or string values, but notice the error that shows up when you enter the following into the interactive shell:

```
 >>> print('I am ' + 29 + ' years old.')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    print('I am ' + 29 + ' years old.')
TypeError: can only concatenate str (not "int") to str
```

The `print()` function isn't causing that error; rather, it's the expression you tried to pass to `print()`. You'll get the same error message if you type the expression into the interactive shell on its own:

```
>>> 'I am ' + 29 + ' years old.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    'I am ' + 29 + ' years old.'
TypeError: can only concatenate str (not "int") to str
```

Python gives an error because the + operator can be used only to add two numbers together or to concatenate two strings. You can't add an integer to a string because this is ungrammatical in Python. You can fix this by using a string version of the integer instead, as explained in the next section.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The str(), int(), and float() Functions

If you want to concatenate an integer such as 29 with a string to pass to print(), you'll need to get the value '29', which is the string form of 29. The str() function can be passed an integer value and will return a string value version of the integer, as follows:

```
>>> str(29)
'29'
>>> print('I am ' + str(29) + ' years old.')
I am 29 years old.
```

Because str(29) evaluates to '29', the expression 'I am ' + str(29) + ' years old.' evaluates to 'I am ' + '29' + ' years old.', which in turn evaluates to 'I am 29 years old.' This is the string value that is passed to the print() function.

The str(), int(), and float() functions will evaluate to the string, integer, and floating-point forms of the value you pass, respectively. Try converting some values in the interactive shell with these functions, and watch what happens:

```
>>> str(0)
'0'
>>> str(-3.14)
'-3.14'
>>> int('42')
42
>>> int('-99')
-99
>>> int(1.25)
1
>>> int(1.99)
1
>>> float('3.14')
3.14
>>> float(10)
10.0
```

The previous examples call the str(), int(), and float() functions and pass them values of the other data types to obtain a string, integer, or floating-point form of those values.

The str() function is handy when you have an integer or float that you want to concatenate to a string. The int() function is also helpful if you have a number as a string value that you want to use in some mathematics. For example, the input() function always returns a string, even if the user enters a number. Enter **spam = input()** into the interactive shell, then enter **101** when it waits for your text:

```
>>> spam = input()
101
>>> spam
'101'
```

The value stored inside spam isn't the integer 101 but the string '101'. If you want to do math using the value in spam, use the int() function to get its integer form and then store this as the variable's new value. If spam is the string '101', then the expression int(spam) will evaluate to the integer value 101, and the assignment statement spam = int(spam) will be equivalent to spam = 101:

```
>>> spam = int(spam)
>>> spam
101
```

Now you should be able to treat the spam variable as an integer instead of a string:

```
>>> spam * 10 / 5
202.0
```

Note that if you pass a value to int() that it cannot evaluate as an integer, Python will display an error message:

```
>>> int('99.99')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    int('99.99')
ValueError: invalid literal for int() with base 10: '99.99'
>>> int('twelve')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    int('twelve')
ValueError: invalid literal for int() with base 10: 'twelve'
```

The int() function is also useful if you need to round a floating-point number down:

```
>>> int(7.7)
7
>>> int(7.7) + 1
8
```

You used the int() and str() functions in the last three lines of your program to get a value of the appropriate data type for the code:

```
print('What is your age?')  # Ask for their age.
my_age = input('>')
print('You will be ' + str(int(my_age) + 1) + ' in a year.')
```

The my_age variable contains the value returned from input(). Because the input() function always returns a string (even if the user entered a number), you can use the int(my_age) code to return an integer value of the string in my_age. This integer value is then added to 1 in the expression int(my_age) + 1.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

TEXT AND NUMBER EQUIVALENCE

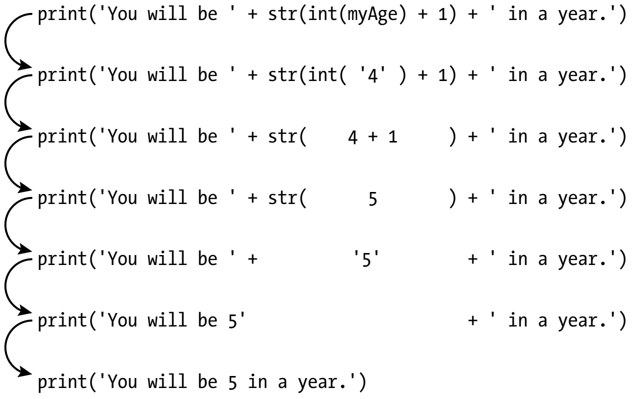
Although the string value of a number is considered a completely different value from the integer or floating-point version, an integer can be equal to a floating point:

```
>>> 42 == '42'
False
>>> 42 == 42.0
True
>>> 42.0 == 0042.000
True
```

Python makes this distinction because strings are text, while integers and floats are numbers.

The result of this addition is passed to the str() function: str(int(my_age) + 1). The string value returned is then concatenated with the strings 'You will be ' and ' in a year.' to evaluate to one large string value. This large string is finally passed to print() to be displayed on the screen.

Let's say the user enters the string '4' for my_age. The evaluation steps would look something like the following:



```
print('You will be ' + str(int(myAge) + 1) + ' in a year.')

print('You will be ' + str(int( '4' ) + 1) + ' in a year.')

print('You will be ' + str(    4 + 1     ) + ' in a year.')

print('You will be ' + str(      5       ) + ' in a year.')

print('You will be ' +        '5'        + ' in a year.')

print('You will be 5'                    + ' in a year.')

print('You will be 5 in a year.')
```

The string '4' is converted to an integer, so you can add 1 to it. The result is 5. The str() function converts the result back to a string, so you can concatenate it with the second string, 'in a year.', to create the final message.

The type() Function

Integer, floating-point, and string aren't the only data types in Python. As you continue to learn about programming, you may come across values of other data types. You can always pass these to the type() function to

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

determine what type they are. For example, enter the following into the interactive shell:

```
>>> type(42)
<class 'int'>
>>> type(42.0)
<class 'float'>
>>> type('forty two')
<class 'str'>
>>> name = 'Zophie'
>>> type(name)  # The name variable has a value of the string type.
<class 'str'>
>>> type(len(name))  # The len() function returns integer values.
<class 'int'>
```

Not only can you pass any value to `type()`, but (as with any function call) you can also pass it any variable or expression to determine the data type of the value that it evaluates to. The `type()` function itself returns values, but the angle brackets mean they are not syntactically valid Python code; you cannot run code like `spam = <class 'str'>`.

The round() and abs() Functions

Let's learn about two more Python functions that, like the `len()` function, take an argument and return a value. The `round()` function accepts a float value and returns the nearest integer. Enter the following into the interactive shell:

```
>>> round(3.14)
3
>>> round(7.7)
8
>>> round(-2.2)
-2
```

The `round()` function also accepts an optional second argument specifying how many decimal places it should round. Enter the following into the interactive shell:

```
>>> round(3.14, 1)
3.1
>>> round(7.7777, 3)
7.778
```

The behavior for rounding half numbers is a bit odd. The function call `round(3.5)` rounds up to `4`, while `round(2.5)` rounds down to `2`. This quirk is caused by how computers handle floating-point numbers and is beyond the scope of this book, but you should keep this behavior in mind.

The `abs()` function returns the absolute value of the number argument. In mathematics, this is defined as the distance from `0`, but I find it easier to think of it as the positive form of the number. Enter the following into the interactive shell:

```
>>> abs(25)
25
>>> abs(-25)
25
>>> abs(-3.14)
3.14
>>> abs(0)
0
```

Python comes with several different functions that you'll learn about in this book. This section should have shown you how you can experiment with them in the interactive shell to see how they behave with different inputs. This is a common technique for practicing the new code that you learn.

How Computers Store Data with Binary Numbers

That's enough Python code for now. At this point, you might think that programming seems almost magical. How does the computer know to transform `2 + 2` into `4`? The answer is too complicated for this book, but I can explain part of what's going on behind the scenes by discussing what binary numbers (numbers that have only the digits 1 and 0) have to do with computing.

Hacking in movies often involves streams of 1s and 0s flowing across the screen. This looks mysterious and impressive, but what do these 1s and 0s actually mean? The answer is that binary is the simplest number system, and it can be implemented with inexpensive components for computer hardware. Binary, also called the *base-2 number system*, can represent all of the same numbers that our more familiar base-10 *decimal number system* can. Decimal has 10 digits, 0 through 9. Table 1-4 shows the first 27 integers in decimal and binary.

Table 1-4: Equivalent Decimal and Binary Numbers

Decimal	Binary	Decimal	Binary	Decimal	Binary
0	0	9	1001	18	10010
1	1	10	1010	19	10011
2	10	11	1011	20	10100
3	11	12	1100	21	10101
4	100	13	1101	22	10110
5	101	14	1110	23	10111
6	110	15	1111	24	11000
7	111	16	10000	25	11001
8	1000	17	10001	26	11010

Think of these number systems as a mechanical odometer, like in Figure 1-3. When you reach the last digit, they each reset to 0 while incrementing the value of the next digit over. In decimal, the last digit is 9, and in binary, the last digit is 1. That's why the decimal number after 9 is 10 and the decimal number after 999 is 1000. Similarly, the binary number after 1 is 10 and the binary number after 111 is 1000. However, *10* in binary doesn't represent the same quantity as *ten* in decimal; rather, it represents *two*. And *1000* in binary doesn't mean *one thousand* in decimal, but rather *eight*. You can view an interactive binary and decimal odometer at *https://inventwithpython.com/odometer*.

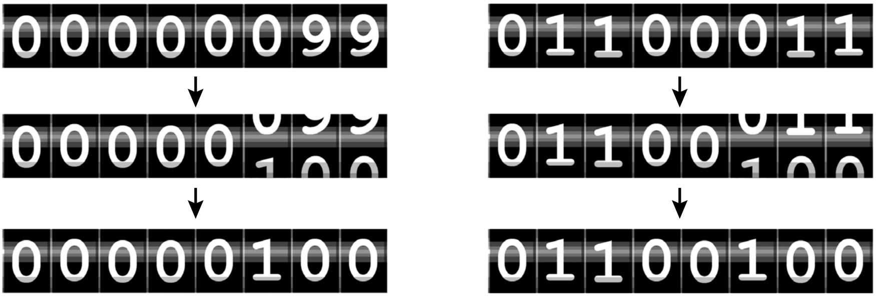


Figure 1-3: A mechanical odometer in decimal (left) and in binary (right)

Representing binary numbers with computer hardware is simpler than representing decimal numbers because there are only two states to represent. For example, Blu-ray discs and DVDs have smooth *lands* and indented *pits* etched on their surface that will or won't reflect the disc player's laser, respectively. Circuits can have electric current flowing through them or no electric current. These various hardware standards all have ways of representing two different states. On the other hand, it'd be expensive to create high-quality electronic components that are sensitive enough to detect the difference between 10 different voltage levels with reliable accuracy. It's more economical to use simple components, and two binary states are as simple as you can get.

These binary digits are called *bits* for short. A single bit can represent two numbers, and 8 bits (or 1 *byte*) can represent 2^8, or 256, numbers, ranging from 0 to 255 in decimal or 0 to 11111111 in binary. This is similar to how a single decimal digit can represent 10 numbers (0 to 9), while an eight-digit decimal number can represent 10^8 or 100,000,000 numbers (0 to 99,999,999). Files on your computer are measured in how many bytes they take up:

- A kilobyte (KB) is 2^{10} or 1,024 bytes.
- A megabyte (MB) is 2^{20} or 1,048,576 bytes (or 1,024KB).
- A gigabyte (GB) is 2^{30} or 1,073,741,824 bytes (or 1,024MB).
- A terabyte (TB) is 2^{40} or 1,099,511,627,776 bytes (or 1,024GB).

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The text of Shakespeare's *Romeo and Juliet* is about 135KB. A high-resolution photo is about 2MB to 5MB. A movie can be anywhere from 1GB to 50GB, depending on picture quality and movie length. However, hard drive and flash memory manufacturers blatantly lie about what these terms mean. For example, by calling a TB 1,000,000,000,000 bytes instead of 1,099,511,627,776 bytes, they can advertise a 9.09TB hard drive as 10TB.

The 1s and 0s of binary can represent not only any integer but also any form of data. Instead of 0 to 255, a byte can represent the numbers –128 to 127 using a system called *two's complement*. Fractional floating-point numbers can be represented in binary using a system called *IEEE-754*.

Text can be stored on computers as binary numbers by assigning each letter, punctuation mark, or symbol a unique number. A system for representing text as numbers is called an *encoding*. The most popular encoding for text is UTF-8. In UTF-8, a capital letter *A* is represented by the decimal number 65 (or 01000001 as an 8-bit binary number), a *?* (question mark) is represented by the number 63, and the numeral character *7* is represented by the number 55. The string `'Hello'` is stored as the numbers 72, 101, 108, 108, and 111. When stored in a computer, "Hello" appears as a stream of bits: 0100000101100101011011000110110001101111.

Wow! Just like in those hacker movies!

Engineers need to invent a way to encode each form of data as numbers. Photos and images can be broken up into a two-dimensional grid of colored squares called *pixels*. Each pixel can use three bytes to represent how much red, green, and blue color it contains. (Chapter 21 covers image data in more detail.) But for a short example, the numbers 255, 0, and 255 could represent a pixel with the maximum amount of red and blue but zero green, resulting in a purple pixel.

Sound is made up of waves of compressed air that reach our ears, which our brains interpret as audio sensation. We can graph the intensity and frequency of these waves over time. The numbers on this graph can then be converted to binary numbers and stored on a computer, which later control speakers to reproduce the sound. This is a simplification of how computer audio works, but describes how numbers can represent Beethoven's Symphony No. 5.

The data for several images combines with audio data to store videos. All forms of information can be encoded into binary numbers. There is, of course, a great deal more detail to it than this, but this is how 1s and 0s represent the wide variety of data in our information age.

Summary

You can compute expressions with a calculator or enter string concatenations with a word processor. You can even do string replication easily by copying and pasting text. But expressions, and their component values—operators, variables, and function calls—are the basic building blocks that make up programs. Once you know how to handle these elements, you will be able to instruct Python to operate on large amounts of data for you.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You'll find it helpful to remember the different types of operators (+, -, *, /, //, %, and ** for math operations, and + and * for string operations) and the three data types (integers, floating-point numbers, and strings) introduced in this chapter.

I introduced a few different functions as well. The print() and input() functions handle simple text output (to the screen) and input (from the keyboard). The len() function takes a string and evaluates to an int of the number of characters in the string. The str(), int(), and float() functions will evaluate to the string, integer, or floating-point number form of the value they are passed. The round() function returns the rounded integer, and the abs() function returns the absolute value of the arguments.

In the next chapter, you'll learn how to tell Python to make intelligent decisions about what code to run, what code to skip, and what code to repeat based on the values it has. This is known as *flow control*, and it allows you to write programs that make intelligent decisions.

Practice Questions

1. Which of the following are operators, and which are values?

```
*
'hello'
-88.8
-
/
+
5
```

2. Which of the following is a variable, and which is a string?

```
spam
'spam'
```

3. Name three data types.
4. What is an expression made up of? What do all expressions do?
5. This chapter introduced assignment statements, like spam = 10. What is the difference between an expression and a statement?
6. What does the variable bacon contain after the following code runs?

```
bacon = 20
bacon + 1
```

7. What should the following two expressions evaluate to?

```
'spam' + 'spamspam'
'spam' * 3
```

8. Why is eggs a valid variable name while 100 is invalid?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

9. What three functions can be used to get the integer, floating-point number, or string version of a value?
10. Why does this expression cause an error? How can you fix it?

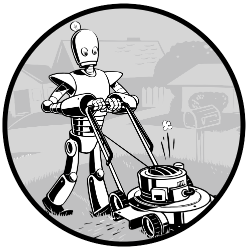
```
'I eat ' + 99 + ' burritos.'
```

Extra credit: Search online for the Python documentation for the `len()` function. It will be on a web page titled "Built-in Functions." Skim the list of other functions Python has, look up what the `bin()` and `hex()` functions do, and experiment with them in the interactive shell.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

2

IF-ELSE AND FLOW CONTROL



So, you know the basics of individual instructions and that a program is just a series of such instructions. But programming's real strength isn't just running one instruction after another like a weekend errand list. Based on how expressions evaluate, a program can decide to skip instructions, repeat them, or choose one of several instructions to run. In fact, you almost never want your programs to start from the first line of code and simply execute every line, straight to the end. *Flow control statements* can decide which Python instructions to execute under which conditions.

These flow control statements directly correspond to the symbols in a flowchart, so I'll provide flowchart versions of the code discussed in this chapter. Figure 2-1 shows a flowchart for what to do if it's raining. Follow the path made by the arrows from Start to End.

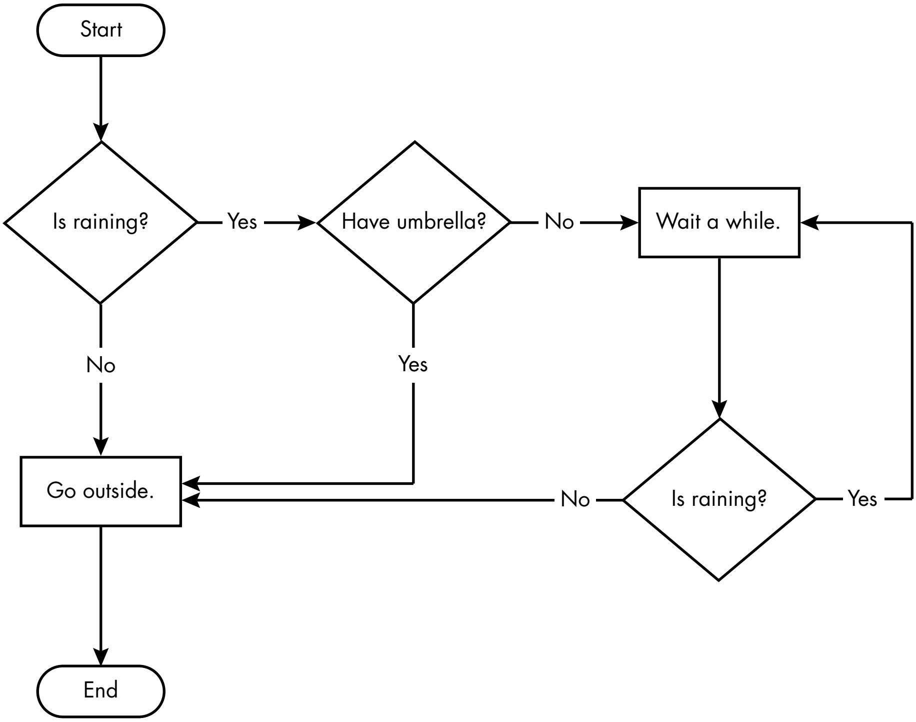


Figure 2-1: A flowchart to tell you what to do if it is raining

In a flowchart, there is usually more than one way to go from the start to the end. The same is true for lines of code in a computer program. Flowcharts represent these branching points with diamonds, the other steps with rectangles, and the starting and ending steps with rounded rectangles.

But before you learn about flow control statements, you first need to learn how to represent those *yes* and *no* options, and you need to understand how to write those branching points as Python code. To that end, let's explore Boolean values, comparison operators, and Boolean operators.

Boolean Values

While the integer, floating-point, and string data types have an unlimited number of possible values, the *Boolean* data type has only two values: `True` and `False`. (*Boolean* is capitalized because the data type is named after mathematician George Boole.) When entered as Python code, the Boolean values `True` and `False` lack the quotation marks you place around strings, and they always start with a capital *T* or *F*, with the rest of the word in lowercase. Note that these Boolean values don't have quotes, because they are

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

different from the string values `'True'` and `'False'`. Enter the following into the interactive shell:

```
❶ >>> spam = True
  >>> spam
  True
❷ >>> true
  Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
  NameError: name 'true' is not defined
❸ >>> False = 2 + 2
    File "<stdin>", line 1, in <module>
  SyntaxError: can't assign to False
```

Some of these instructions are intentionally incorrect, and they'll cause error messages to appear. Like any other value, you can use Boolean values in expressions and store them in variables ❶. If you don't use the proper case ❷ or if you try to use `True` and `False` for variable names ❸, Python will give you an error message.

Comparison Operators

Comparison operators, also called *relational operators*, compare two values and evaluate down to a single Boolean value. Table 2-1 lists the comparison operators.

Table 2-1: Comparison Operators

Operator	Meaning	Examples
`==`	Equal to	`5 == 5` evaluates to `True`. `4 == 2 + 2` evaluates to `True`.
`!=`	Not equal to	`1  != 2` evaluates to `True`. `'Hello'  != 'Hello'` evaluates to `False`.
`<`	Less than	`10 < 5` evaluates to `False`. `1.999 < 5` evaluates to `True`.
`>`	Greater than	`1 + 1 > 4 + 8` evaluates to `False`. `99 > 4 + 8` evaluates to `True`.
`<=`	Less than or equal to	`4 <= 5` evaluates to `True`. `5 <=5` evaluates to `True`.
`>=`	Greater than or equal to	`5 >= 4` evaluates to `True`. `5 >= 5` evaluates to `True`.

These operators evaluate to `True` or `False` depending on the values you give them. Let's try some operators now, starting with `==` and `!=`:

```
>>> 42 == 42
True
>>> 42 == 99
False
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> 2 != 3
True
>>> 2 != 2
False
```

As you might expect, == (equal to) evaluates to True when the values on both sides are the same, and != (not equal to) evaluates to True when the two values are different. The == and != operators can actually work with values of any data type:

```
>>> 'hello' == 'hello'
True
>>> 'hello' == 'Hello'
False
>>> 'dog' != 'cat'
True
>>> True == True
True
>>> True != False
True
>>> 42 == 42.0
True
❶ >>> 42 == '42'
False
```

Note that an integer or floating-point value will never equal a string value. The expression 42 == '42' ❶ evaluates to False because Python considers the integer 42 to be different from the string '42'. However, Python does consider the integer 42 to be the same as the float 42.0.

The <, >, <=, and >= operators, on the other hand, work properly only with integer and floating-point values:

```
>>> 42 < 100
True
>>> 42 > 100
False
>>> 42 < 42
False
>>> eggs = 42
❶ >>> eggs <= 42
True
>>> my_age = 29
❷ >>> my_age >= 10
True
```

You'll often use comparison operators to compare a variable's value to some other value, like in the eggs <= 42 ❶ and my_age >= 10 ❷ examples, or to compare the values in two variables to each other. (After all, comparing two literal values like 'dog' != 'cat' always has the same result.) You'll see more examples of this later when you learn about flow control statements.

THE DIFFERENCE BETWEEN THE == AND = OPERATORS

You might have noticed that the == operator (equal to) has two equal signs, while the = operator (assignment) has just one equal sign. It's easy to confuse these two operators with each other. Just remember these points:

- The == operator asks whether two values are the same as each other.
- The = operator puts the value on the right into the variable on the left.

To help remember which is which, notice that the == operator (equal to) consists of two characters, just like the != operator (not equal to) consists of two characters.

Boolean Operators

The three Boolean operators (and, or, and not) are used to compare Boolean values. Like comparison operators, they evaluate these expressions down to a Boolean value. Let's explore these operators in detail, starting with the and operator.

The and operator always takes two Boolean values (or expressions), so it's considered to be a *binary* Boolean operator. The and operator evaluates an expression to True if *both* Boolean values are True; otherwise, it evaluates to False. Enter some expressions using and into the interactive shell to see it in action:

```
>>> True and True
True
>>> True and False
False
```

A *truth table* shows every possible result of a Boolean operator. Table 2-2 is the truth table for the and operator.

Table 2-2: The and Operator's Truth Table

Expression	Evaluates to . . .
True and True	True
True and False	False
False and True	False
False and False	False

Like the and operator, the or operator also always takes two Boolean values (or expressions), and therefore is considered to be a binary Boolean operator. However, the `or` operator evaluates an expression to `True` if *either* of the two Boolean values is `True`. If both are `False`, it evaluates to `False`:

```
>>> False or True
True
>>> False or False
False
```

You can see every possible outcome of the `or` operator in its truth table, shown in Table 2-3.

Table 2-3: The `or` Operator's Truth Table

Expression	Evaluates to . . .
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Unlike `and` and `or`, the `not` operator operates on only one Boolean value (or expression). This makes it a *unary* operator. The `not` operator simply evaluates to the opposite Boolean value:

```
>>> not True
False
❶ >>> not not not not True
True
```

Much like using double negatives in speech and writing, you can use multiple `not` operators ❶, though there's never not no reason to do this in real programs. Table 2-4 shows the truth table for `not`.

Table 2-4: The `not` Operator's Truth Table

Expression	Evaluates to . . .
`not True`	`False`
`not False`	`True`

Mixing Boolean and Comparison Operators

Since the comparison operators evaluate to Boolean values, you can use them in expressions with the Boolean operators.

Recall that the `and`, `or`, and `not` operators are called Boolean operators because they always operate on the Boolean values `True` and `False`. While expressions like `4 < 5` aren't Boolean values, they are expressions that

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

evaluate down to Boolean values. Try entering some Boolean expressions that use comparison operators into the interactive shell:

```
>>> (4 < 5) and (5 < 6)
True
>>> (4 < 5) and (9 < 6)
False
>>> (1 == 2) or (2 == 2)
True
```

The computer will evaluate the left expression first, and then it will evaluate the right expression. When it knows the Boolean value for each, it will evaluate the whole expression down to one Boolean value. You can think of the computer's evaluation process for `(4 < 5) and (5 < 6)` as the following:

```
(4 < 5) and (5 < 6)
         ↓
  True and (5 < 6)
         ↓
   True and True
         ↓
        True
```

You can also use multiple Boolean operators in an expression, along with the comparison operators:

```
>>> spam = 4
>>> 2 + 2 == spam and not 2 + 2 == (spam + 1) and 2 * 2 == 2 + 2
True
```

The Boolean operators have an order of operations just like the math operators do. After any math and comparison operators evaluate, Python evaluates the `not` operators first, then the `and` operators, and then the `or` operators.

Components of Flow Control

Flow control statements often start with a part called the *condition* and are always followed by a block of code called the *clause*. Before you learn about Python's specific flow control statements, I'll cover what a condition and a block are.

Conditions

The Boolean expressions you've seen so far could all be considered conditions, which are the same thing as expressions; *condition* is just a more specific name in the context of flow control statements. A condition always evaluates to a Boolean value, `True` or `False`. A flow control statement decides what to do based on whether its condition is `True` or `False`, and almost every flow control statement uses a condition. You'll frequently write code that

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

could be described in English as follows: "If this condition is true, do this thing, or else do this other thing." Other code you'll write is the same as saying, "Keep repeating these instructions as long as this condition continues to be true."

Blocks of Code

Lines of Python code can be grouped together in *blocks*. You can tell when a block begins and ends from the indentation of the lines of code. There are four rules for blocks:

- A new block begins when the indentation increases.
- Blocks can contain other blocks.
- A block ends when the indentation decreases to zero or to a containing block's indentation.
- Python expects a new block immediately after any statement that ends with a colon.

Blocks are easier to understand by looking at some indented code, so let's find the blocks in part of a small program, shown here:

```
username = 'Mary'
password = 'swordfish'
if username == 'Mary':
  ❶ print('Hello, Mary')
    if password == 'swordfish':
      ❷ print('Access granted.')
    else:
      ❸ print('Wrong password.')
```

The first block of code ❶ starts at the line `print('Hello, Mary')` and contains all the lines after it. Inside this block is another block ❷, which has only a single line in it: `print('Access granted.')`. The third block ❸ is also one line long: `print('Wrong password.')`.

Program Execution

In the previous chapter's *hello.py* program, Python started executing instructions at the top of the program going down, one after another. *Program execution* (or simply, *execution*) is a term for the current instruction being executed. If you put your finger on each line on your screen as the line is executed, you can think of your finger as the program execution.

Not all programs execute by simply going straight down, however. If you use your finger to trace through a program with flow control statements, you'll likely find your finger jumping around to different places in the source code based on conditions.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Flow Control Statements

Now let's explore the most important piece of flow control: the statements themselves. The statements represent the diamonds you saw in the flowchart in Figure 2-1, and they are the actual decisions your programs will make.

if

The most common type of flow control statement is the `if` statement. An `if` statement's clause (that is, the block following the `if` statement) will execute if the statement's condition is `True`. The clause is skipped if the condition is `False`.

In plain English, an `if` statement could be read as, "If this condition is true, execute the code in the clause." In Python, an `if` statement consists of the following:

- The `if` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon
- Starting on the next line, an indented block of code (called the `if` clause or `if` block)

For example, let's say you have some code that checks whether someone's name is Alice:

```
name = 'Alice'
if name == 'Alice':
    print('Hi, Alice.')
```

All flow control statements end with a colon and are followed by a new block of code (the clause). This `if` statement's clause is the block with `print('Hi, Alice.')`. Figure 2-2 shows what the flowchart of this code would look like.

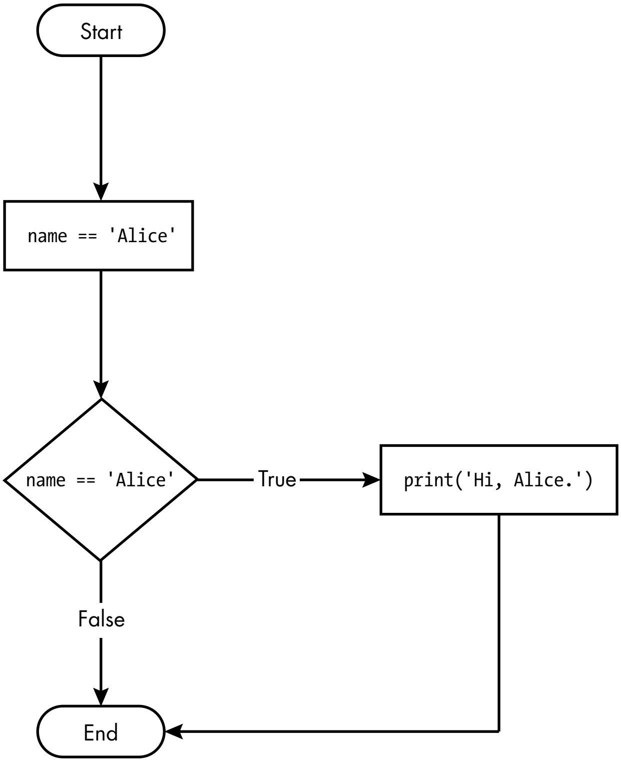


Figure 2-2: The flowchart for an `if` statement

Try changing the `name` variable to another string besides `'Alice'`, and run the program again. Notice that "Hi, Alice." doesn't appear on the screen, because that code was skipped over.

else

An `if` clause can optionally be followed by an `else` statement. The `else` clause is executed only when the `if` statement's condition is `False`. In plain English, an `else` statement could be read as, "If this condition is true, execute this code. Or else, execute that code." An `else` statement doesn't have a condition, and in code, an `else` statement always consists of the following:

- The `else` keyword
- A colon
- Starting on the next line, an indented block of code (called the `else` clause or `else` block)

Returning to the Alice example, let's look at some code that uses an `else` statement to offer a different greeting if the person's name isn't Alice:

```
name = 'Alice'
if name == 'Alice':
    print('Hi, Alice.')
else:
    print('Hello, stranger.')
```

Figure 2-3 shows what the flowchart of this code would look like.

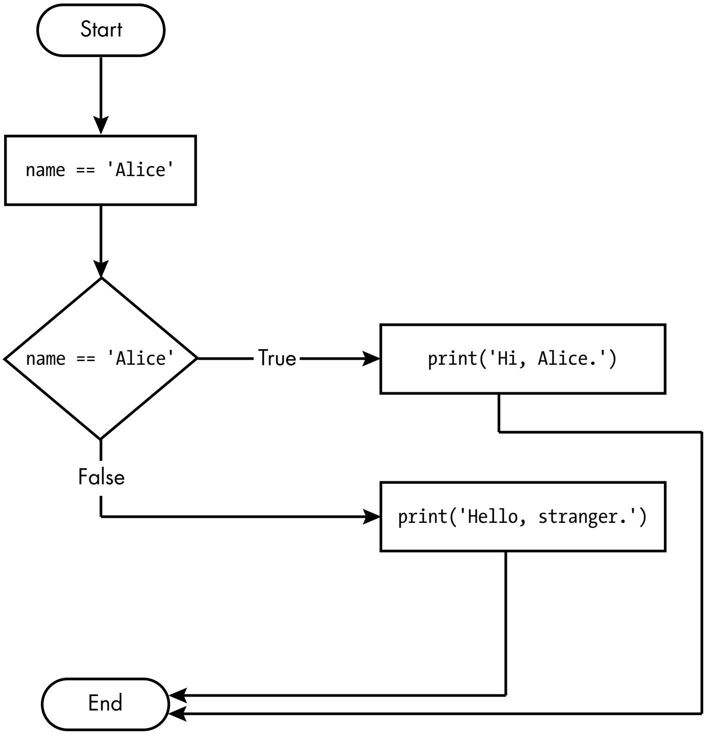


Figure 2-3: The flowchart for an `else` *statement*

Try changing the name variable to a string besides `'Alice'`, and rerun the program. Instead of `'Hi, Alice.'`, you will see `'Hello, stranger.'` on the screen.

elif

You would use `if` or `else` when you want only one of the clauses to execute. But you may have a case where you want one of *many* possible clauses to execute. The `elif` statement is an "else if" statement that always follows an `if` or another `elif` statement. It provides another condition that is checked only if all of the previous conditions were `False`. In code, an `elif` statement always consists of the following:

- The `elif` keyword
- A condition (that is, an expression that evaluates to `True` or `False`)
- A colon

- Starting on the next line, an indented block of code (called the `elif` clause or `elif` block)

Let's add an `elif` to the name checker to see this statement in action:

```
name = 'Alice'
age = 33
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
```

This time, the program checks the person's age and tells them something different if they're younger than 12. You can see the corresponding flowchart in Figure 2-4.

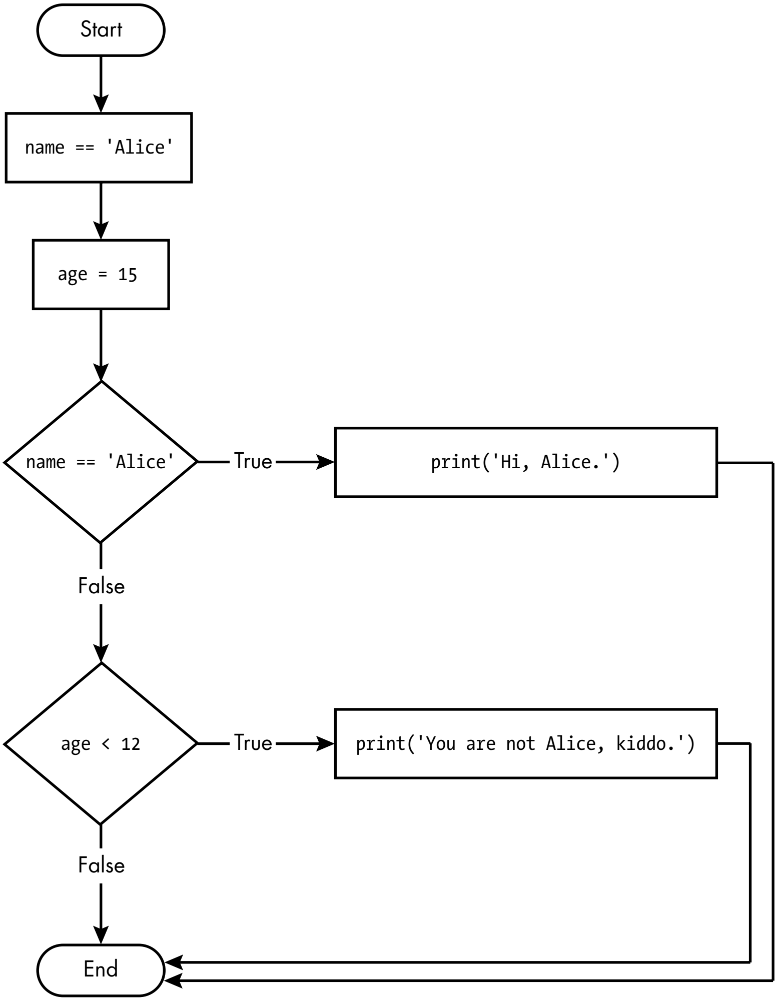


Figure 2-4: The flowchart for an `elif` *statement*

The elif clause executes if age < 12 is True and name == 'Alice' is False. However, if both of the conditions are False, Python skips both of the clauses. There is *no* guarantee that at least one of the clauses will be executed; in a chain of elif statements, only one or none of the clauses will be executed. Once one of the statements' conditions is found to be True, the rest of the elif clauses are automatically skipped. For example, open a new file editor window and enter the following code, saving it as *vampire.py*:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
elif age > 2000:
    print('Unlike you, Alice is not an undead, immortal vampire.')
elif age > 100:
    print('You are not Alice, grannie.')
```

Here, I've added two more elif statements to make the name checker greet a person with different answers based on age. Figure 2-5 shows the flowchart for this.

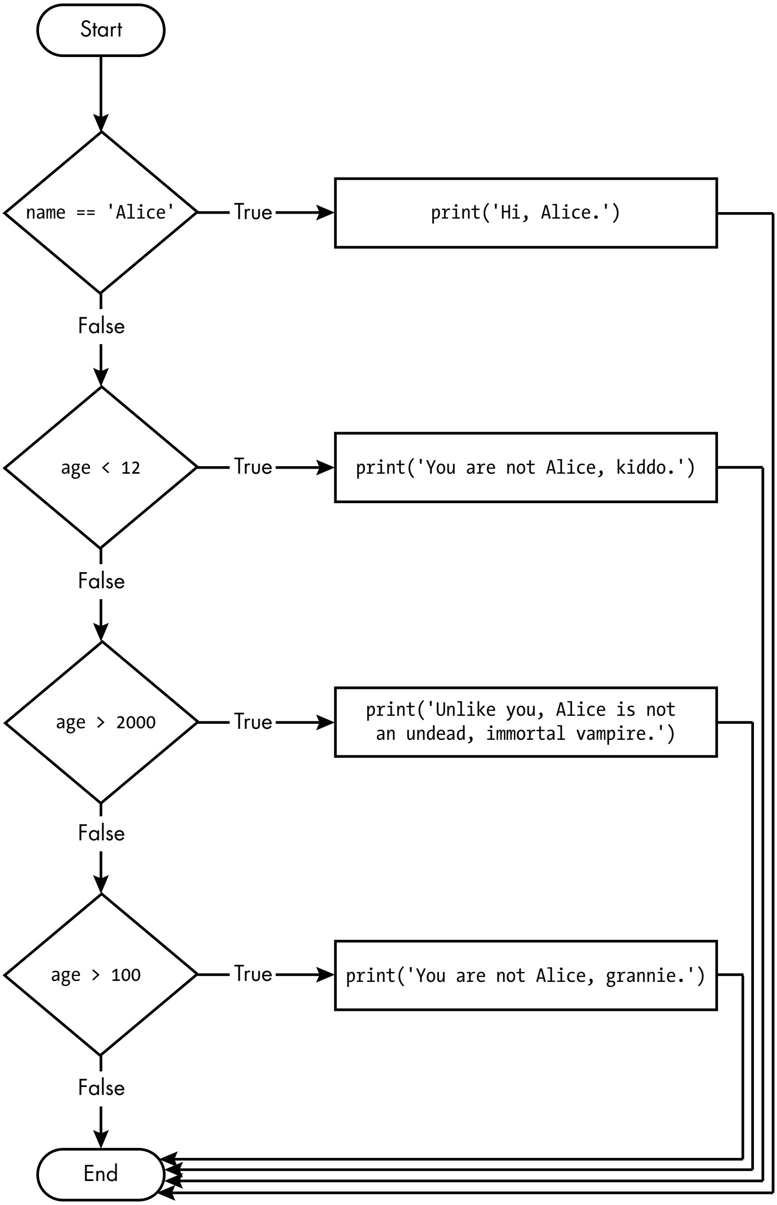

Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
age > 100
True
print('You are not Alice, grannie.')
False
End

Figure 2-5: The flowchart for multiple `elif` *statements in the* vampire.py *program*

The order of the `elif` statements does matter, however. Let's rearrange them to introduce a bug. Remember that the rest of the `elif` clauses are automatically skipped once a `True` condition has been found, so if you swap around some of the clauses in *vampire.py*, you will run into a problem. Change the code to look like the following, and save it as *vampire2.py*:

```
  name = 'Carol'
  age = 3000
  if name == 'Alice':
      print('Hi, Alice.')
  elif age < 12:
      print('You are not Alice, kiddo.')
❶ elif age > 100:
      print('You are not Alice, grannie.')
  elif age > 2000:
      print('Unlike you, Alice is not an undead, immortal vampire.')
```

Say the `age` variable contains the value `3000` before this code is executed. You might expect the code to print the string `'Unlike you, Alice is not an undead, immortal vampire.'` However, because the `age > 100` condition is `True` (after all, 3,000 *is* greater than 100) ❶, the string `'You are not Alice, grannie.'` is printed, and the rest of the `elif` statements are automatically skipped. Remember that at most only one of the clauses will be executed, and for `elif` statements, the order matters!

Figure 2-6 shows the flowchart for the previous code. Notice how the diamonds for `age > 100` and `age > 2000` are swapped.

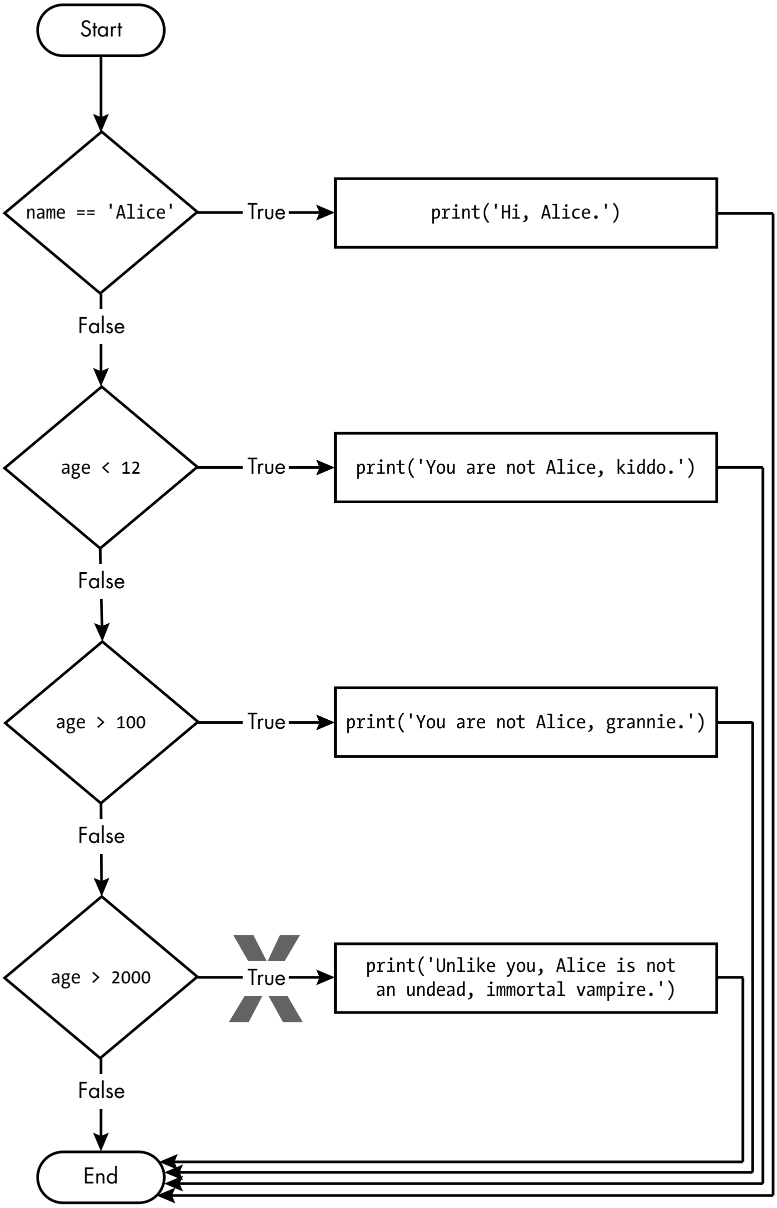

Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
age > 100
True
print('You are not Alice, grannie.')
False
age > 2000
True
print('Unlike you, Alice is not an undead, immortal vampire.')
False
End

Figure 2-6: The flowchart for the vampire2.py *program. The X path will logically never happen, because if* `age` *were greater than* `2000`*, it would have already been greater than* `100`*.*

Optionally, you can have an `else` statement after the last `elif` statement. In that case, it *is* guaranteed that at least one (and only one) of the clauses will be executed. If the conditions in every `if` and `elif` statement are `False`, then the else clause is executed. For example, let's re-create the Alice program to use if, elif, and else clauses:

```
name = 'Carol'
age = 3000
if name == 'Alice':
    print('Hi, Alice.')
elif age < 12:
    print('You are not Alice, kiddo.')
else:
    print('You are neither Alice nor a little kid.')
```

Figure 2-7 shows the flowchart for this new code, which we'll save as *littleKid.py*.

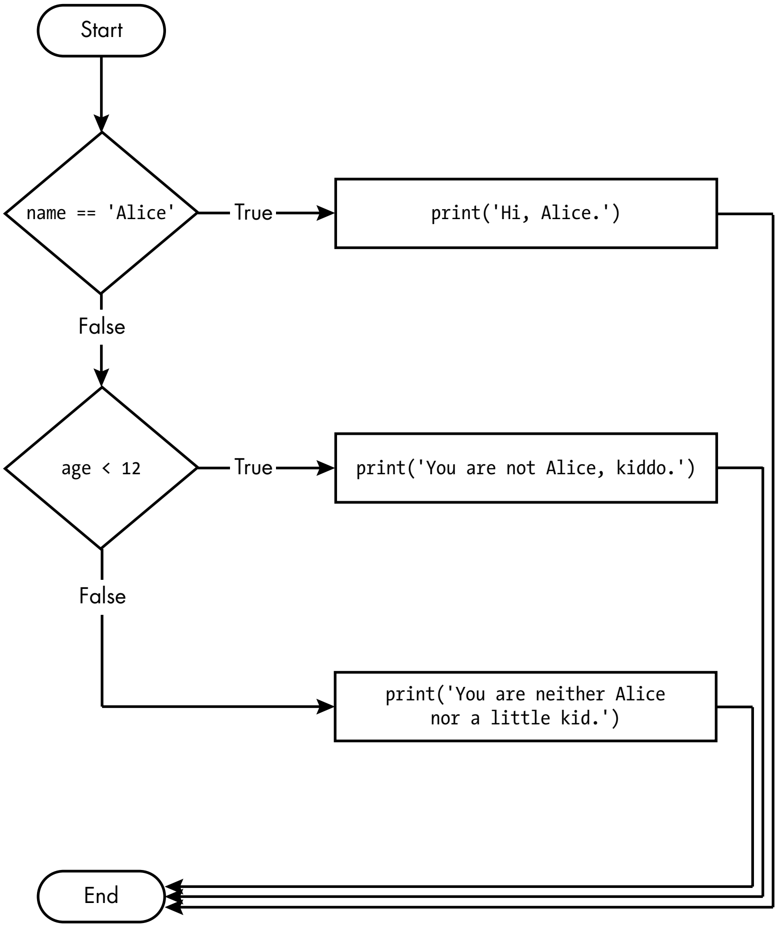

Start
name == 'Alice'
True
print('Hi, Alice.')
False
age < 12
True
print('You are not Alice, kiddo.')
False
print('You are neither Alice nor a little kid.')
End

Figure 2-7: The flowchart for the littleKid.py *program*

In plain English, this type of flow control structure would be, "If the first condition is true, do this. Else, if the second condition is true, do that.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Otherwise, do something else.” When you use `if`, `elif`, and `else` statements together, remember these rules about how to order them to avoid bugs like the one in Figure 2-6. First, there is always exactly one `if` statement; any `elif` statements you need should follow the `if` statement. Second, if you want to be sure that at least one clause is executed, close the structure with an `else` statement.

As you can see, flow control statements can make your programs more sophisticated but also more complicated. Don’t despair; you will become more comfortable with this complexity as you practice writing code. And all true programmers have at some point spent an hour to find out their program doesn’t work because they accidentally typed `<` instead of `<=`. These little mistakes happen to everyone.

A Short Program: Opposite Day

With your knowledge of Boolean values and `if`-`else` statements, enter the following code into a new file and save it as *oppositeday.py*:

```
today_is_opposite_day = True

# Set say_it_is_opposite_day based on today_is_opposite_day:
❶ if today_is_opposite_day == True:
    say_it_is_opposite_day = True
else:
    say_it_is_opposite_day = False

# If it is opposite day, toggle say_it_is_opposite_day:
if today_is_opposite_day == True:
  ❷ say_it_is_opposite_day = not say_it_is_opposite_day

# Say what day it is:
if say_it_is_opposite_day == True:
    print('Today is Opposite Day.')
else:
    print('Today is not Opposite Day.')
```

When you run this program, it outputs `'Today is not Opposite Day.'` There are two variables in this code. At the start of the program, the `today_is_opposite_day` variable is set to `True`. The next `if` statement checks if this variable is `True` (it is) ❶ and sets the `say_it_is_opposite_day` variable to `True`; otherwise, it would set the variable to `False`. The second `if` statement checks if `today_is_opposite_day` is set to `True` (it still is), and if so, the code *toggles* (that is, sets to the opposite Boolean value) the variable ❷. Finally, the third `if` statement checks if `say_it_is_opposite_day` is `True` (it isn’t), and prints `'Today is Opposite Day.'`; otherwise, it would have printed `'Today is not Opposite Day.'`

If you change the first line of the program to `today_is_opposite_day = False` and run the program again, it still prints `'Today is not Opposite Day.'` If we look through the program, we can figure out that the first `if`-`else` statements set `say_it_is_opposite_day` to `False`. The second `if` statement’s

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

condition is False, so it skips its block of code. Finally, the third if statement's condition is again False and prints 'Today is not Opposite Day.'

So, if today is not Opposite Day, the program correctly prints 'Today is not Opposite Day.' And if today is Opposite Day, the program (also correctly) prints 'Today is not Opposite Day.' as one would say on Opposite Day. Logically, this program will never print 'Today is Opposite Day.' no matter if the variable is set to True or False. Really, you could replace this entire program with just print('Today is not Opposite Day.') and it would be the same program. This is why programmers should not be paid per line of code written.

A Short Program: Dishonest Capacity Calculator

In Chapter 1, I showed how hard drive and flash memory manufacturers lie about the advertised capacities of their products by using a different definition of TB and GB. Let's write a program to calculate how misleading their advertised capacities are. Enter the following code into a new file and save it as *dishonestcapacity.py*:

```
print('Enter TB or GB for the advertised unit:')
unit = input('>')

# Calculate the amount that the advertised capacity lies:
if unit == 'TB' or unit == 'tb':
    discrepancy = 1000000000000 / 1099511627776
elif unit == 'GB' or unit == 'gb':
    discrepancy = 1000000000 / 1073741824

print('Enter the advertised capacity:')
advertised_capacity = input('>')
advertised_capacity = float(advertised_capacity)

# Calculate the real capacity, round it to the nearest hundredths,
# and convert it to a string so it can be concatenated:
real_capacity = str(round(advertised_capacity * discrepancy, 2))

print('The actual capacity is ' + real_capacity + ' ' + unit)
```

This program asks the user to enter what unit the hard drive advertises itself as having, either TB or GB:

```
# Calculate the amount that the advertised capacity lies:
if unit == 'TB' or unit == 'tb':
    discrepancy = 1000000000000 / 1099511627776
elif unit == 'GB' or unit == 'gb':
    discrepancy = 1000000000 / 1073741824
```

TBs are larger than GBs, and the larger the unit, the wider the discrepancy between advertised and real capacities. The if and elif statements use a Boolean or operator so that the program works no matter whether the

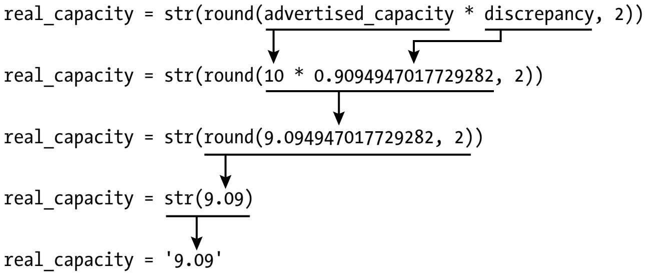
Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

user enters the unit in lowercase or uppercase. If the user enters something else for the unit, then neither the `if` clause nor the `elif` clause runs, and the discrepancy variable is never assigned. Later, when the program tries to use the discrepancy variable, this will cause an error. We'll cover that case later.

Next, the user enters the advertised size in the given units:

```
# Calculate the real capacity, round it to the nearest hundredths,
# and convert it to a string so it can be concatenated:
real_capacity = str(round(advertised_capacity * discrepancy, 2))
```

We do a lot in this single line. Let's use the example of the user entering 10TB for the advertised size and unit. If we look at the innermost part of the line of code, we see that `advertised_capacity` is multiplied by `discrepancy`. This is the real capacity, but it may have several digits, as in `9.094947017729282`. So this number is passed as the first argument to `round()` with `2` as the second argument. This function call to `round()` returns, in our example, `9.09`. This is a floating-point value, but we want to get a string form of it to concatenate to a message string in the next line of code. To do this, we pass it to the `str()` function. Python evaluates this one line as the following:



```
real_capacity = str(round(advertised_capacity * discrepancy, 2))

real_capacity = str(round(10 * 0.9094947017729282, 2))

real_capacity = str(round(9.094947017729282, 2))

real_capacity = str(9.09)

real_capacity = '9.09'
```

If the user failed to enter *TB*, *tb*, *GB*, or *gb* as the unit, the conditions for both the `if` and `elif` statements would be `False` and the `discrepancy` variable would never be created. But the user wouldn't know anything was wrong until Python tried to use the nonexistent variable. Python would raise a `NameError: name 'discrepancy' is not defined` error and point to the line where `real_capacity` is assigned.

The true origin of this bug, however, is the fact that the program doesn't handle the case where the user enters an invalid unit. There are many ways to handle this error, but the simplest would be to have an `else` clause that displays a message like "You must enter TB or GB" and then calls the `sys.exit()` function to quit the program. (You'll learn about this function in the next chapter.)

The final line in the program displays the actual hard drive capacity by concatenating a message string to the `real_capacity` and `unit` strings:

```
print('The actual capacity is ' + real_capacity + ' ' + unit)
```

As it turns out, hard drives and flash memory manufacturers lie even more: I have a 256GB SD card in my laptop that I use for backups. In real

GBs, this should be 274,877,906,944 bytes. In fake GBs, it should be 256,000,000,000 bytes. But my computer reports that the actual capacity is 255,802,212,352 bytes. It's funny how the actual size is always inaccurate in a way that makes it less than the advertised size, and never more.

Summary

By using expressions that evaluate to `True` or `False` (also called conditions), you can write programs that make decisions on what code to execute and what code to skip. These conditions are expressions that compare two values with the `==`, `!=`, `<`, `>`, `<=`, and `>=` comparison operators to evaluate to a Boolean value. You can also use the `and`, `or`, and `not` Boolean operators to connect expressions into more complicated expressions. Python uses indentation to create blocks of code. In this chapter, we used blocks as part of `if`, `elif`, and `else` statements, but as you'll see, several other Python statements use blocks as well. These flow control statements will let you write more intelligent programs.

Practice Questions

1. What are the two values of the Boolean data type? How do you write them?
2. What are the three Boolean operators?
3. Write out the truth tables of each Boolean operator (that is, every possible combination of Boolean values for the operator and what they evaluate to).
4. What do the following expressions evaluate to?

```
(5 > 4) and (3 == 5)
not (5 > 4)
(5 > 4) or (3==5)
not ((5 > 4) or (3==5))
(True and True) and (True == False)
(not False) or (not True)
```

5. What are the six comparison operators?
6. What is the difference between the equal to operator and the assignment operator?
7. Explain what a condition is and where you would use one.
8. Identify the three blocks in this code:

```
spam = 0
if spam == 10:
    print('eggs')
    if spam > 5:
        print('bacon')
```

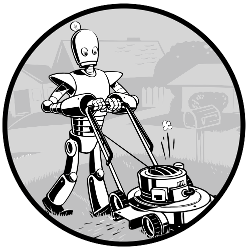
Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    else:
        print('ham')
    print('spam')
print('spam')
```

9. Write code that prints `Hello` if `1` is stored in `spam`, prints `Howdy` if `2` is stored in `spam`, and prints `Greetings!` if anything else is stored in `spam`.

3

LOOPS



In the previous chapter, you learned how to make programs run certain blocks of code while skipping others. But there's more to flow control than this. In this chapter, you'll learn how to repeatedly execute blocks of code using loops. Python's two kinds of loops, `while` and `for`, open up the full power of automation, because they can run lines of code millions of times per second. You'll also learn how to import code libraries, called *modules*, to make even more functions available to your programs.

while Loop Statements

You can make a block of code execute over and over again using a `while` statement. The code in a `while` clause will be executed as long as the statement's condition is True. In code, a while statement always consists of the following:

- The while keyword
- A condition (that is, an expression that evaluates to True or False)
- A colon
- Starting on the next line, an indented block of code (called the while clause or while block)

You can see that a while statement looks similar to an if statement. The difference is in how they behave. At the end of an if clause, the program execution continues after the if statement. But at the end of a while clause, the program execution jumps back to the start of the while statement. The while clause is often called the *while loop* or just the *loop*.

Let's look at an if statement and a while loop that use the same condition and take the same actions based on that condition. Here is the code with an if statement:

```
spam = 0
if spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

Here is the code with a while statement:

```
spam = 0
while spam < 5:
    print('Hello, world.')
    spam = spam + 1
```

These statements are similar; both if and while check the value of spam, and if it's less than 5, they print a message. But when you run these two code snippets, something very different happens for each one. For the if statement, the output is simply "Hello, world." But for the while statement, it's "Hello, world." repeated five times! Take a look at the flowcharts for these two pieces of code, Figures 3-1 and 3-2, to see why this happens.

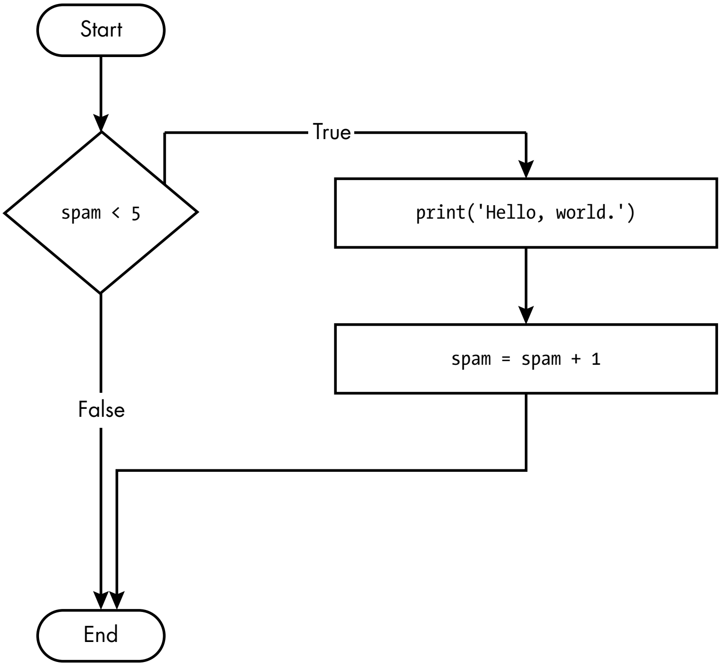


Figure 3-1: The flowchart for the `if` statement code

The code with the `if` statement checks the condition, and it prints `"Hello, world."` only once if that condition is true. The code with the `while` loop, on the other hand, will print it five times. The loop stops after five prints because the integer in `spam` increases by one at the end of each loop iteration, which means that the loop will execute five times before `spam < 5` is `False`.

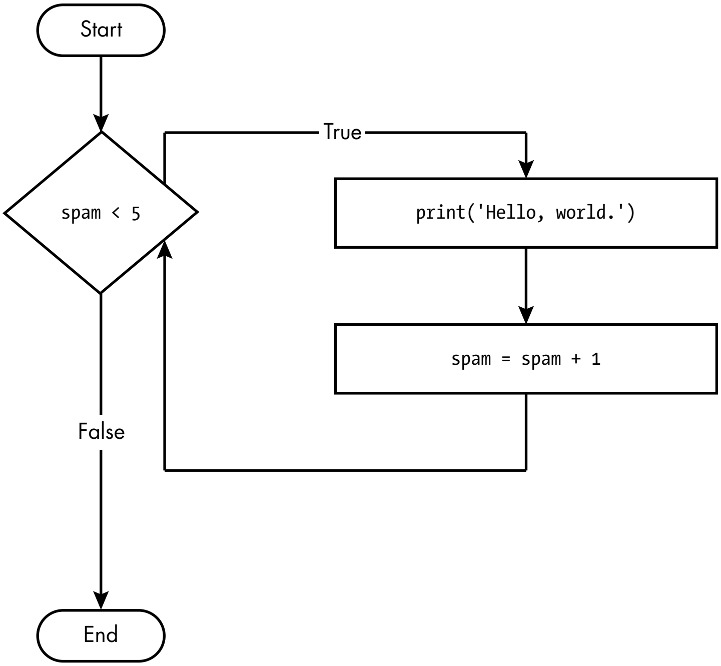


Figure 3-2: The flowchart for the `while` *statement code*

In the `while` loop, the condition is always checked at the start of each *iteration* (that is, each time the loop is executed). If the condition is `True`, then the clause is executed, and afterward, the condition is checked again. The first time the condition is found to be `False`, the `while` clause is skipped.

An Annoying while Loop

Here's a small example program that will keep asking you to type, literally, *your name*. Select **File ▸ New** to open a new file editor window, enter the following code, and save the file as *yourName.py*:

```
name = ''
while name != 'your name':
    print('Please type your name.')
    name = input('>')
print('Thank you!')
```

First, the program sets the `name` variable to an empty string. This is so that the `name != 'your name'` condition will evaluate to `True` and the program execution will enter the `while` loop's clause.

The code inside this clause asks the user to type their name, which is assigned to the `name` variable. Since this is the last line of the block, the execution moves back to the start of the `while` loop and reevaluates the condition. If the value in `name` is *not* equal to the string `'your name'`, the condition is `True`, and the execution enters the `while` clause again.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

But once the user literally enters *your name*, the condition of the `while` loop will be `'your name' != 'your name'`, which evaluates to `False`. Now, instead of the program execution reentering the `while` loop's clause, Python skips past it and continues running the rest of the program. Figure 3-3 shows the flowchart for the *yourName.py* program.

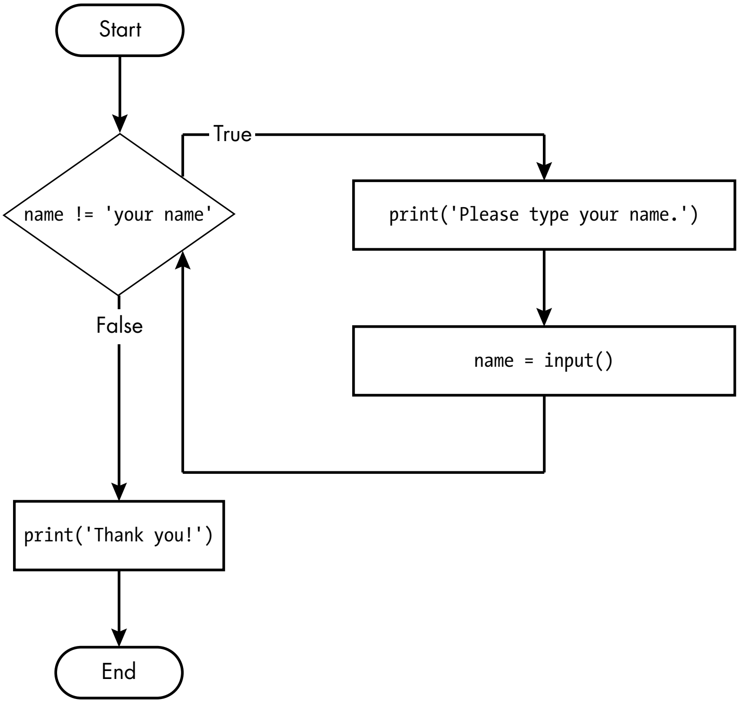


Figure 3-3: The flowchart for the yourName.py *program*

Now let's see *yourName.py* in action. Press F5 to run it, and enter something other than *your name* a few times before you give the program what it wants:

```
Please type your name.
>Al
Please type your name.
>Albert
Please type your name.
>%#@#%*(^&!!!
Please type your name.
>your name
Thank you!
```

If you never enter *your name*, then the `while` loop's condition will never be `False`, and the program will just keep asking you the question forever. Here, the `input()` call lets the user enter the right string to make the program move on. In other programs, the condition might never actually

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

change, and that can be a problem. Let's look at how you can break out of a `while` loop.

break Statements

There is a shortcut to getting the program execution to break out of a `while` loop's clause early. If the execution reaches a `break` statement, it immediately exits the `while` loop's clause. In code, a `break` statement simply contains the `break` keyword.

Here's a program that does the same thing as the previous *yourName.py* program but uses a `break` statement to escape the loop. Enter the following code, and save the file as *yourName2.py*:

```
while True:
    print('Please type your name.')
    name = input('>')
    if name == 'your name':
        break
print('Thank you!')
```

The first line creates an *infinite loop*; it is a `while` loop whose condition is always `True`. (The expression `True`, after all, always evaluates to the value `True`.) After the program execution enters this loop, it will exit the loop only when a `break` statement is executed. (An infinite loop that *never* exits is a common programming bug.)

Just like before, this program asks the user to enter *your name*. Now, however, while the execution is still inside the `while` loop, an `if` statement checks whether `name` is equal to `'your name'`. If this condition is `True`, the `break` statement is run, and the execution moves out of the loop to `print('Thank you!')`. Otherwise, the `if` statement clause that contains the `break` statement is skipped, which puts the execution at the end of the `while` loop. At this point, the program execution jumps back to the start of the `while` statement to recheck the condition. Since this condition is merely the `True` Boolean value, the execution enters the loop to ask the user to enter *your name* again. See Figure 3-4 for this program's flowchart.

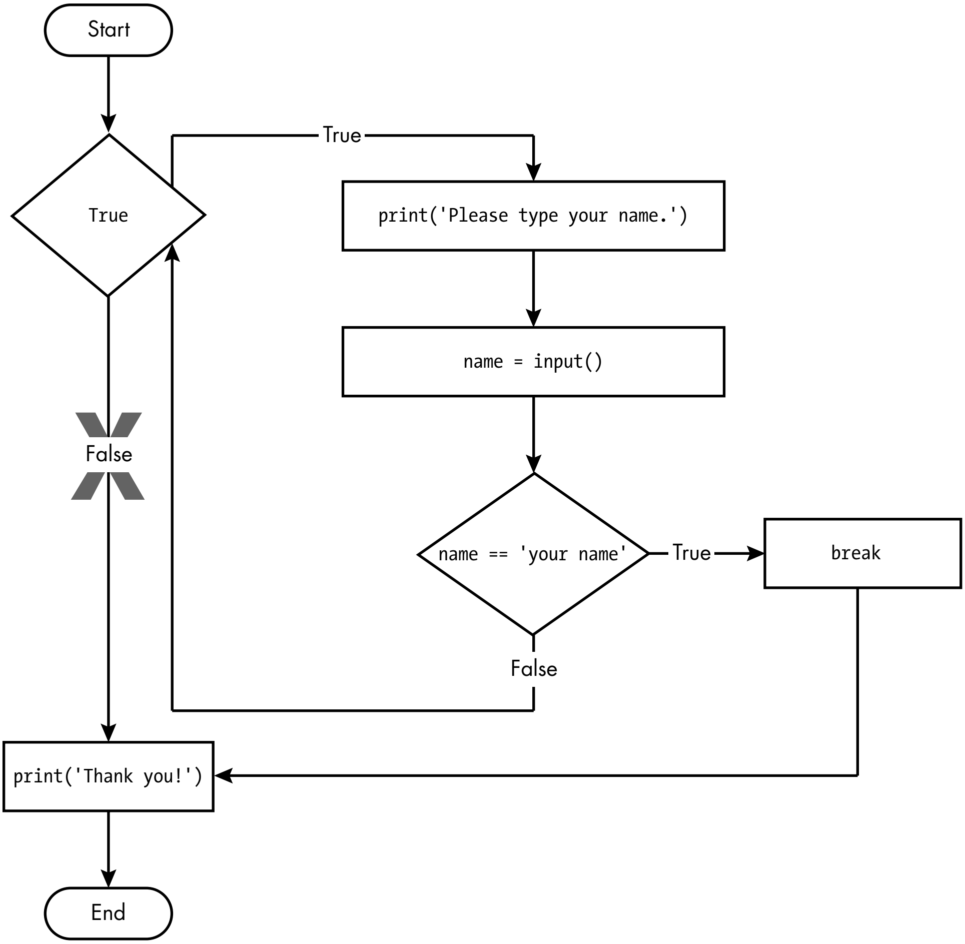


Figure 3-4: The flowchart for the yourName2.py *program with an infinite loop. Note that the X path will logically never happen, because the loop condition is always* `True`*.*

Run *yourName2.py*, and enter the same text you entered for *yourName.py*. The rewritten program should respond in the same way as the original.

continue Statements

Like `break` statements, we use `continue` statements inside loops. When the program execution reaches a `continue` statement, the program execution immediately jumps back to the start of the loop and reevaluates the loop's condition. (This is also what happens when the execution reaches the end of the loop.)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

TRAPPED IN AN INFINITE LOOP?

If you ever run a program that has a bug causing it to get stuck in an infinite loop, press CTRL-C. This will send a KeyboardInterrupt error to your program and cause it to stop immediately. You can also click the Stop button at the top of the Mu window. Try stopping a program by creating a simple infinite loop in the file editor, and save the program as *infiniteLoop.py*. If you are running the program from Mu, you can also click the Stop button:

```
while True:
    print('Hello, world!')
```

When you run this program, it will print Hello, world! to the screen forever because the while statement's condition is always True. CTRL-C is also handy if you want to simply terminate your program immediately, even if it's not stuck in an infinite loop.

Let's use continue to write a program that asks for a name and password. Enter the following code into a new file editor window and save the program as *swordfish.py*:

```
while True:
    print('Who are you?')
    name = input('>')
  ❶ if name != 'Joe':
      ❷ continue
    print('Hello, Joe. What is the password? (It is a fish.)')
  ❸ password = input('>')
    if password == 'swordfish':
      ❹ break
❺ print('Access granted.')
```

If the user enters any name besides Joe ❶, the continue statement ❷ causes the program execution to jump back to the start of the loop. When the program reevaluates the condition, the execution will always enter the loop, because the condition is simply the value True. Once the user makes it past that if statement, they are asked for a password ❸. If the password entered is swordfish, the break statement ❹ is run, and the execution jumps out of the while loop to print Access granted. ❺. Otherwise, the execution continues to the end of the while loop, where it then jumps back to the start of the loop. See Figure 3-5 for this program's flowchart.

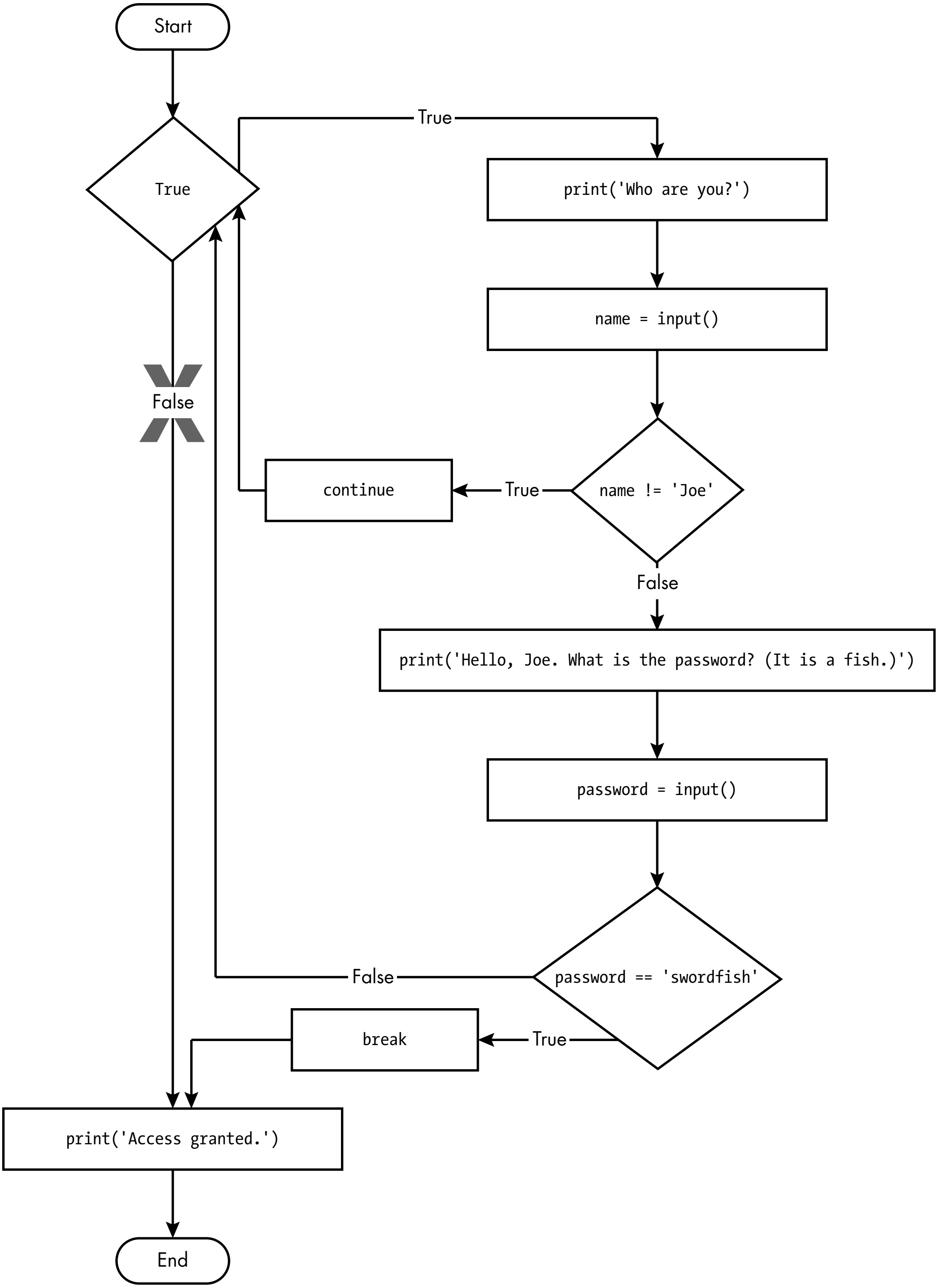


Figure 3-5: The flowchart for the swordfish.py *program. The X path will logically never happen, because the loop condition is always* `True`*.*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Run this program and give it some input. Until you claim to be Joe, the program shouldn't ask for a password, and once you enter the correct password, it should exit:

```
Who are you?
>I'm fine, thanks. Who are you?
Who are you?
>Joe
Hello, Joe. What is the password? (It is a fish.)
>Mary
Who are you?
>Joe
Hello, Joe. What is the password? (It is a fish.)
>swordfish
Access granted.
```

"TRUTHY" AND "FALSEY" VALUES AND THE BOOL() FUNCTION

Conditions will consider some values in other data types equivalent to `True` or `False`. When used in conditions, `0`, `0.0`, and `''` (the empty string) are considered `False`, while all other values are considered `True`. For example, look at the following program:

```
  name = ''
❶ while not name:
      print('Enter your name:')
      name = input('>')
  print('How many guests will you have?')
  numOfGuests = int(input('>'))
❷ if numOfGuests:
    3 print('Be sure to have enough room for all your guests.')
  print('Done')
```

If the user enters a blank string for `name`, the `while` statement's condition will be `True` ❶, and the program will continue to ask for a name. If the value for `numOfGuests` is not 0 ❷, the condition is considered to be `True`, and the program will print a reminder for the user ❸.

You could have entered `not name != ''` instead of `not name`, and `numOfGuests != 0` instead of `numOfGuests`, but using the truthy and falsey values can make your code easier to read.

If you want to know if a value is truthy or falsey, pass it to the `bool()` function as in this interactive shell example:

```
>>> bool(0)
False
```

(continued)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> bool(42)
True
>>> bool('Hello')
True
>>> bool('')
False
```

for Loops and the range() Function

The `while` loop keeps looping while its condition is `True` (which is the reason for its name), but what if you want to execute a block of code only a certain number of times? You can do this with a `for` loop statement and the `range()` function.

In code, a `for` statement looks something like `for i in range(5):` and includes the following:

- The `for` keyword
- A variable name
- The `in` keyword
- A call to the `range()` function with up to three integers passed to it
- A colon
- Starting on the next line, an indented block of code (called the `for` clause or `for` block)

Let's create a new program called *fiveTimes.py* to help you see a `for` loop in action:

```
print('Hello!')
for i in range(5):
    print('On this iteration, i is set to ' + str(i))
print('Goodbye!')
```

The code in the `for` loop's clause is run five times. The first time it is run, the variable `i` is set to `0`. The `print()` call in the clause will print `On this iteration, i is set to 0`. After Python finishes an iteration through all the code inside the `for` loop's clause, the execution goes back to the top of the loop, and the `for` statement increments `i` by one. This is why `range(5)` results in five iterations through the clause, with `i` being set to `0`, then `1`, then `2`, then `3`, and then `4`. The variable `i` will go up to, but will not include, the integer passed to `range()`. Figure 3-6 shows the flowchart for the *fiveTimes.py* program.

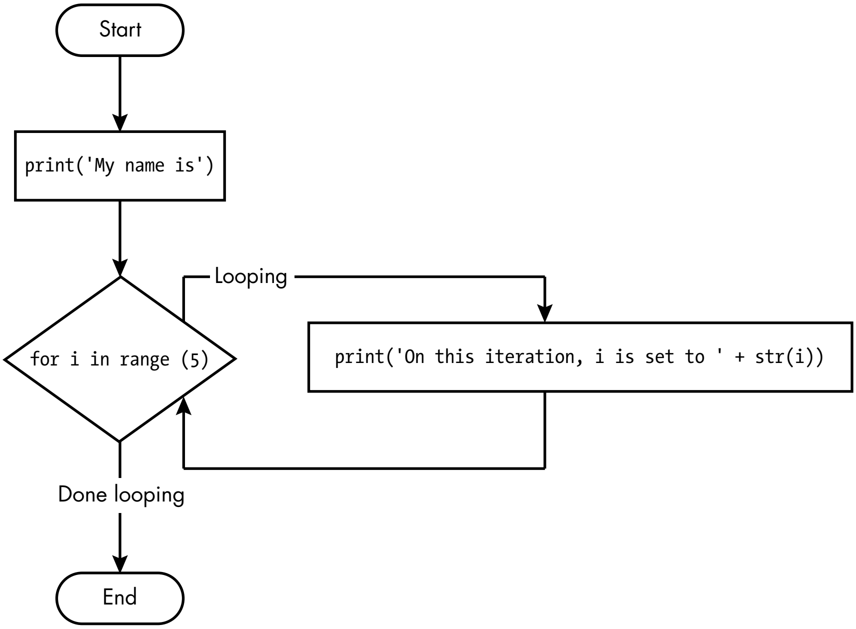


Figure 3-6: The flowchart for the fiveTimes.py *program*

Here is the complete output of the program:

```
Hello!
On this iteration, i is set to 0
On this iteration, i is set to 1
On this iteration, i is set to 2
On this iteration, i is set to 3
On this iteration, i is set to 4

Goodbye!
```

You can use `break` and `continue` statements inside `for` loops as well. The `continue` statement will continue to the next value of the `for` loop's counter, as if the program execution had reached the end of the loop and returned to the start. In fact, you can use `continue` and `break` statements only inside `while` and `for` loops. If you try to use these statements elsewhere, Python will give you an error.

As another `for` loop example, consider this story about the mathematician Carl Friedrich Gauss. When Gauss was a boy, a teacher wanted to give the class some busywork. The teacher told them to add up all the numbers from 0 to 100. Young Gauss came up with a clever trick to figure out the answer in a few seconds, but you can write a Python program with a `for` loop to do this calculation for you:

```
total = 0
for num in range(101):
    total = total + num
print(total)
```

The result should be 5,050. When the program first starts, the `total` variable is set to `0`. The `for` loop then executes `total = total + num` 101 times, each time with an incremented `i` variable. By the time the loop has finished all of its 101 iterations, every integer from `0` to `100` will have been added to `total`. At this point, `total` is printed to the screen. Even on the slowest computers, this program takes less than a second to complete.

(Young Gauss figured out a way to solve the problem in seconds. There are 50 pairs of numbers that add up to 101: 1 + 100, 2 + 99, 3 + 98, and so on, until 50 + 51. Since 50 × 101 is 5,050, the sum of all the numbers from 0 to 100 is 5,050. Clever kid!)

WHY "UP TO BUT NOT INCLUDING"?

It may seem strange that the `range()` function in `for` loops has you specify the number that the loop goes up to, minus one. In programming, ranges are often specified in a "closed, open" format that includes the starting number but excludes the ending number. For example, the range 0, 10 would include the numbers 0 to 9 instead of 0 to 10. There are many advantages of using "closed, open" instead of "closed, closed" that lead to fewer bugs. Calculating the size of the range is just a matter of subtracting the starting number from the ending number. The 0, 10 range (which has the numbers 0 to 9 instead of 0 to 10) has 10 - 0 or 10 numbers. With a "closed, closed" range of 0, 9, you have to calculate the length as 9 - 0 + 1. (And it's easy to mistakenly do *off-by-one* errors like 9 - 0 - 1.) The start of the next range 10, 20 is just a matter of using the previous ending number as the new starting number.

For programs that deal with timestamps, the "closed, open" range of 00:00:00, 24:00:00.0 is much easier to work with than the "closed, closed" range of 00:00:00, 23:59:59.999. "Closed, open" ranges may seem odd at first, but they'll become second nature as you get more experience writing code.

An Equivalent while Loop

You can actually use a `while` loop to do the same thing as a `for` loop; `for` loops are just more concise. Let's rewrite *fiveTimes.py* to use a `while` loop equivalent of a `for` loop:

```
print('Hello!')
i = 0
while i < 5:
    print('On this iteration, i is set to ' + str(i))
    i = i + 1
print('Goodbye!')
```

If you run this program, the output should look the same as the *five Times.py* program, which uses a `for` loop. Remember that `for` loops are

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

useful for looping a specific number of times, and while loops are useful for looping as long as a particular condition is true.

Arguments to range()

Some functions can be called with multiple arguments separated by a comma, and range() is one of them. This lets you change the integer passed to range() to follow any sequence of integers, including starting at a number other than zero:

```
for i in range(12, 16):
    print(i)
```

The first argument will be where the for loop's variable starts, and the second argument will be up to, but not including, the number to stop at:

```
12
13
14
15
```

The range() function can also be called with three arguments. The first two arguments will be the start and stop values, and the third will be the *step argument*. The step is the amount by which the variable is increased after each iteration:

```
for i in range(0, 10, 2):
    print(i)
```

So, calling range(0, 10, 2) will count from zero to eight by intervals of two:

```
0
2
4
6
8
```

The range() function is flexible in the sequence of numbers it produces for for loops. *For* example (I never apologize for my puns), you can even use a negative number for the step argument to make the for loop count down instead of up:

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

This for loop would have the following output:

```
5
4
3
2
```

```
1
0
```

Running a `for` loop to print `i` with `range(5, -1, -1)` should print the numbers from five down to zero.

Importing Modules

All Python programs can call a basic set of functions called *built-in functions*, including the `print()`, `input()`, and `len()` functions you've seen before. Python also comes with a set of modules called the *standard library*. Each module is a Python program that contains a related group of functions that can be embedded in your programs. For example, the `math` module has mathematics-related functions, the `random` module has random number–related functions, and so on.

Before you can use the functions in a module, you must import the module with an `import` statement. In code, an `import` statement consists of the following:

- The `import` keyword
- The name of the module
- Optionally, more module names, as long as they are separated by commas

Once you import a module, you can use all the cool functions of that module. Let's give it a try with the `random` module, which will give us access to the `random.randint()` function.

Enter this code into the file editor, and save it as *printRandom.py*:

```
import random
for i in range(5):
    print(random.randint(1, 10))
```

When you run this program, the output will look something like this:

```
4
1
8
4
1
```

The `random.randint()` function call evaluates to a random integer value between the two integers that you pass it. Because `randint()` is in the `random` module, you must first enter **`random.`** in front of the function name to tell Python to look for this function inside the `random` module.

DON'T OVERWRITE MODULE NAMES

When you save your Python scripts, take care not to give them a name that is used by one of Python's modules, such as *random.py*, *sys.py*, *os.py*, or *math.py*. If you accidentally name one of your programs, say, *random.py*, and use an `import random` statement in another program, your program will import your *random.py* file instead of Python's `random` module. This can lead to errors such as `AttributeError: module 'random' has no attribute 'randint'`, since your *random.py* file doesn't have the functions that the real `random` module has. Don't use the names of any built-in Python functions for your file or variable names, either. Some common Python names are `all`, `any`, `date`, `email`, `file`, `format`, `hash`, `id`, `input`, `list`, `min`, `max`, `object`, `open`, `random`, `set`, `str`, `sum`, `test`, and `type`.

Here's an example of an `import` statement that imports four different modules:

```
import random, sys, os, math
```

Now we can use any of the functions in these four modules. You'll learn more about them later in the book.

An alternative form of the `import` statement is composed of the `from` keyword, followed by the module name, the `import` keyword, and a star (*); for example, `from random import *`. With this form of `import` statement, calls to functions in `random` won't need the `random.` prefix. However, using the full name makes for more readable code, so it is better to use the `import random` form of the statement.

Ending a Program Early with sys.exit()

The last flow control concept to cover is how to *terminate*, or exit, the program. Programs always terminate if the program execution reaches the bottom of the instructions. However, you can cause the program to terminate before the last instruction by calling the `sys.exit()` function. Since this function is in the sys module, you have to import sys before your program can use it.

Open a file editor window and enter the following code, saving it as *exitExample.py*:

```
import sys

while True:
    print('Type exit to exit.')
    response = input('>')
    if response == 'exit':
        sys.exit()
    print('You typed ' + response + '.')
```

Run this program in your code editor. This program has an infinite loop with no `break` statement inside. The only way this program will end is if the execution reaches the `sys.exit()` call. When `response` is equal to `'exit'`, the line containing the `sys.exit()` call is executed. Since the `response` variable is set by the `input()` function, the user must enter `exit` in order to stop the program.

A Short Program: Guess the Number

The examples I've shown you so far are useful for introducing basic concepts, but now you'll see how everything you've learned comes together in a more complete program. In this section, I'll show you a simple guess the number game. When you run this program, the output will look something like this:

```
I am thinking of a number between 1 and 20.
Take a guess.
>10
Your guess is too low.
Take a guess.
>15
Your guess is too low.
Take a guess.
>17
Your guess is too high.
Take a guess.
>16
Good job! You got it in 4 guesses!
```

Enter the following source code into the file editor, and save the file as *guessTheNumber.py*:

```
# This is a guess the number game.
import random
secret_number = random.randint(1, 20)
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guesses_taken in range(1, 7):
    print('Take a guess.')
    guess = int(input('>'))

    if guess < secret_number:
        print('Your guess is too low.')
    elif guess > secret_number:
        print('Your guess is too high.')
    else:
        break  # This condition is the correct guess!
```

```
if guess == secret_number:
    print('Good job! You got it in ' + str(guesses_taken) + ' guesses!')
else:
    print('Nope. The number was ' + str(secret_number))
```

Let's look at this code line by line, starting at the top:

```
# This is a guess the number game.
import random
secret_number = random.randint(1, 20)
```

First, a comment at the top of the code explains what the program does. Then, the program imports the `random` module so that it can use the `random.randint()` function to generate a number for the user to guess. The return value, a random integer between 1 and 20, is stored in the variable `secret_number`:

```
print('I am thinking of a number between 1 and 20.')

# Ask the player to guess 6 times.
for guesses_taken in range(1, 7):
    print('Take a guess.')
    guess = int(input('>'))
```

The program tells the player that it has come up with a secret number and will give the player six chances to guess it. The code lets the player enter a guess and checks that guess in a `for` loop that will loop at most six times. The first thing that happens in the loop is that the player types in a guess. Because `input()` returns a string, its return value is passed straight into `int()`, which translates the string into an integer value. This gets stored in a variable named `guess`:

```
    if guess < secret_number:
        print('Your guess is too low.')
    elif guess > secret_number:
        print('Your guess is too high.')
```

These few lines of code check whether the guess is less than or greater than the secret number. In either case, a hint is printed to the screen:

```
    else:
        break  # This condition is the correct guess!
```

If the guess is neither higher nor lower than the secret number, then it must be equal to the secret number—in which case, you want the program execution to break out of the `for` loop:

```
if guess == secret_number:
    print('Good job! You got it in ' + str(guesses_taken) + ' guesses!')
else:
    print('Nope. The number was ' + str(secret_number))
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

After the for loop, the previous if-else statement checks whether the player has correctly guessed the number and then prints an appropriate message to the screen. In both cases, the program displays a variable that contains an integer value (guesses_taken and secret_number). Since it must concatenate these integer values to strings, it passes these variables to the str() function, which returns the string value form of these integers. Now these strings can be concatenated with the + operators before finally being passed to the print() function call.

A Short Program: Rock, Paper, Scissors

Let's use the programming concepts we've learned so far to create a simple rock, paper, scissors game. The output will look like this:

```
ROCK, PAPER, SCISSORS
0 Wins, 0 Losses, 0 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
>p
PAPER versus...
PAPER
It is a tie!
0 Wins, 0 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
>s
SCISSORS versus...
PAPER
You win!
1 Wins, 0 Losses, 1 Ties
Enter your move: (r)ock (p)aper (s)cissors or (q)uit
>q
```

Enter the following source code into the file editor, and save the file as *rpsGame.py*:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
losses = 0
ties = 0

while True:  # The main game loop
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True:  # The player input loop
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        player_move = input('>')
        if player_move == 'q':
            sys.exit()  # Quit the program.
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
        if player_move == 'r' or player_move == 'p' or player_move == 's':
            break  # Break out of the player input loop.
        print('Type one of r, p, s, or q.')

    # Display what the player chose:
    if player_move == 'r':
        print('ROCK versus...')
    elif player_move == 'p':
        print('PAPER versus...')
    elif player_move == 's':
        print('SCISSORS versus...')

    # Display what the computer chose:
    move_number = random.randint(1, 3)
    if move_number == 1:
        computer_move = 'r'
        print('ROCK')
    elif move_number == 2:
        computer_move = 'p'
        print('PAPER')
    elif move_number == 3:
        computer_move = 's'
        print('SCISSORS')

    # Display and record the win/loss/tie:
    if player_move == computer_move:
        print('It is a tie!')
        ties = ties + 1
    elif player_move == 'r' and computer_move == 's':
        print('You win!')
        wins = wins + 1
    elif player_move == 'p' and computer_move == 'r':
        print('You win!')
        wins = wins + 1
    elif player_move == 's' and computer_move == 'p':
        print('You win!')
        wins = wins + 1
    elif player_move == 'r' and computer_move == 'p':
        print('You lose!')
        losses = losses + 1
    elif player_move == 'p' and computer_move == 's':
        print('You lose!')
        losses = losses + 1
    elif player_move == 's' and computer_move == 'r':
        print('You lose!')
        losses = losses + 1
```

Let’s look at this code line by line, starting at the top:

```
import random, sys

print('ROCK, PAPER, SCISSORS')

# These variables keep track of the number of wins, losses, and ties.
wins = 0
```

```
losses = 0
ties = 0
```

First, we import the random and sys modules so that our program can call the random.randint() and sys.exit() functions. We also set up three variables to keep track of how many wins, losses, and ties the player has had:

```
while True:  # The main game loop
    print('%s Wins, %s Losses, %s Ties' % (wins, losses, ties))
    while True:  # The player input loop
        print('Enter your move: (r)ock (p)aper (s)cissors or (q)uit')
        player_move = input('>')
        if player_move == 'q':
            sys.exit()  # Quit the program.
        if player_move == 'r' or player_move == 'p' or player_move == 's':
            break  # Break out of the player input loop.
        print('Type one of r, p, s, or q.')
```

This program uses a while loop inside another while loop. The first loop is the main game loop, and a single game of rock, paper, scissors is played on each iteration through this loop. The second loop asks for input from the player, and keeps looping until the player has entered an r, p, s, or q for their move. The r, p, and s correspond to rock, paper, and scissors, respectively, while the q means the player intends to quit. In that case, sys.exit() is called and the program exits. If the player has entered r, p, or s, the execution breaks out of the loop. Otherwise, the program reminds the player to enter r, p, s, or q and goes back to the start of the loop:

```
    # Display what the player chose:
    if player_move == 'r':
        print('ROCK versus...')
    elif player_move == 'p':
        print('PAPER versus...')
    elif player_move == 's':
        print('SCISSORS versus...')
```

The player's move is displayed on the screen:

```
    # Display what the computer chose:
    move_number = random.randint(1, 3)
    if move_number == 1:
        computer_move = 'r'
        print('ROCK')
    elif move_number == 2:
        computer_move = 'p'
        print('PAPER')
    elif move_number == 3:
        computer_move = 's'
        print('SCISSORS')
```

Next, the program randomly selects the computer's move. Because random.randint() will always return a random number, the code stores the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

1, 2, or 3 integer value it returns in a variable named move_number. The program then stores an 'r', 'p', or 's' string in computer_move based on the integer in move_number, as well as displays the computer's move:

```
    # Display and record the win/loss/tie:
    if player_move == computer_move:
        print('It is a tie!')
        ties = ties + 1
    elif player_move == 'r' and computer_move == 's':
        print('You win!')
        wins = wins + 1
    elif player_move == 'p' and computer_move == 'r':
        print('You win!')
        wins = wins + 1
    elif player_move == 's' and computer_move == 'p':
        print('You win!')
        wins = wins + 1
    elif player_move == 'r' and computer_move == 'p':
        print('You lose!')
        losses = losses + 1
    elif player_move == 'p' and computer_move == 's':
        print('You lose!')
        losses = losses + 1
    elif player_move == 's' and computer_move == 'r':
        print('You lose!')
        losses = losses + 1
```

Finally, the program compares the strings in player_move and computer_move, and displays the results on the screen. It also increments the wins, losses, or ties variable appropriately. Once the execution reaches the end, it jumps back to the start of the main program loop to begin another game.

If you liked these guess the number and rock, paper, scissors games, you can find the source code to other simple Python programs in *The Big Book of Small Python Projects* (No Starch Press, 2021).

Summary

Loops let your programs execute code over and over again while a certain condition evaluates to True. The break and continue statements are useful if you need to exit a loop or jump back to the loop's start.

These flow control statements will let you write more intelligent programs. You can also use another type of flow control by writing your own functions, which is the topic of the next chapter.

Practice Questions

1. What keys can you press if your Python program is stuck in an infinite loop?
2. What is the difference between break and continue?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

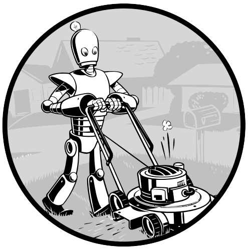
3. What is the difference between `range(10)`, `range(0, 10)`, and `range(0, 10, 1)` in a `for` loop?
4. Write a short program that prints the numbers `1` to `10` using a `for` loop. Then, write an equivalent program that prints the numbers `1` to `10` using a `while` loop.
5. If you had a function named `bacon()` inside a module named `spam`, how would you call it after importing `spam`?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

4

FUNCTIONS



A *function* is like a mini program within a program. Python provides several built-in functions, such as the `print()`, `input()`, and `len()` functions from the previous chapters, but you can also write your own. In this chapter, you'll create functions, explore the call stack used to determine the order in which functions in a program run, and learn about the scope of variables inside and outside functions.

Creating Functions

To better understand how functions work, let's create one. Enter this program into the file editor and save it as *helloFunc.py*:

```
def hello():
    # Prints three greetings
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    print('Good morning!')
    print('Good afternoon!')
    print('Good evening!')

hello()
hello()
print('ONE MORE TIME!')
hello()
```

The first line is a `def` statement, which defines a function named `hello()`. The code in the block that follows the `def` statement is the body of the function. This code executes when the function is called, not when the function is first defined.

The `hello()` lines after the function are function calls. In code, a function call is just the function's name followed by parentheses, possibly with some number of arguments in between the parentheses. When the program execution reaches these calls, it will jump to the first line in the function and begin executing the code there. When it reaches the end of the function, the execution returns to the line that called the function and continues moving through the code as before.

Because this program calls `hello()` three times, the code in the `hello()` function is executed three times. When you run this program, the output looks like this:

```
Good morning!
Good afternoon!
Good evening!
Good morning!
Good afternoon!
Good evening!
ONE MORE TIME!
Good morning!
Good afternoon!
Good evening!
```

A major purpose of functions is to group code that gets executed multiple times. Without a function defined, you would have to copy and paste this code each time you wanted to run it, and the program would look like this:

```
print('Good morning!')
print('Good afternoon!')
print('Good evening!')
print('Good morning!')
print('Good afternoon!')
print('Good evening!')
print('ONE MORE TIME!')
print('Good morning!')
print(Good afternoon!'')
print(' Good evening!')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

In general, you always want to avoid duplicating code, because if you ever decide to update the code (for example, because you find a bug you need to fix), you'll have to remember to change the code in every place you copied it.

As you gain programming experience, you'll often find yourself *deduplicating*, which means getting rid of copied-and-pasted code. Deduplication makes your programs shorter, easier to read, and easier to update.

Arguments and Parameters

When you call the `print()` or `len()` function, you pass it values, called *arguments*, by entering them between the parentheses. You can also define your own functions that accept arguments. Enter this example into the file editor and save it as *helloFunc2.py*:

```
❶ def say_hello_to(name):
      # Prints three greetings to the name provided
    ❷ print('Good morning, ' + name)
      print('Good afternoon, ' + name)
      print('Good evening, ' + name)

❸ say_hello_to('Alice')
  say_hello_to('Bob')
```

When you run this program, the output looks like this:

```
Good morning, Alice
Good afternoon, Alice
Good evening, Alice
Good morning, Bob
Good afternoon, Bob
Good evening, Bob
```

The definition of the `say_hello_to()` function has a parameter called `name` ❶. *Parameters* are variables that contain arguments. When a function is called with arguments, the arguments are stored in the parameters. The first time the `say_hello_to()` function is called, it's passed the argument `'Alice'` ❸. The program execution enters the function, and the parameter `name` is automatically set to `'Alice'`, which gets printed by the `print()` statement ❷. You should use parameters in your function if you need it to follow slightly different instructions depending on the values you pass to the function call.

One special thing to note about parameters is that the value stored in a parameter is forgotten when the function returns. For example, if you added `print(name)` after `say_hello_to('Bob')` in the previous program, the program would give you an error because there is no variable named `name`. This variable gets destroyed after the function call `say_hello_to('Bob')` returns, so `print(name)` would refer to a `name` variable that does not exist.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The terms *define*, *call*, *pass*, *argument*, and *parameter* can be confusing. To review their meanings, consider a code example:

```
❶ def say_hello_to(name):
      # Prints three greetings to the name provided
      print('Good morning, ' + name)
      print('Good afternoon, ' + name)
      print('Good evening, ' + name)
❷ say_hello_to('Al')
```

To *define* a function is to create it, just as an assignment statement like `spam = 42` creates the `spam` variable. The `def` statement defines the `say_hello_to()` function ❶. The `say_hello_to('Al')` line ❷ *calls* the now-created function, sending the execution to the top of the function's code. This function call is also known as *passing* the string value `'Al'` to the function. A value being passed to a function in a function call is an *argument*. The argument `'Al'` is assigned to a local variable named `name`. Variables that have arguments assigned to them are *parameters*.

It's easy to mix up these terms, but keeping them straight will ensure that you know precisely what the text in this chapter means.

Return Values and return Statements

When you call the `len()` function and pass it an argument such as `'Hello'`, the function call evaluates to the integer value `5`, which is the length of the string you passed it. In general, the value to which a function call evaluates is called the *return value* of the function.

When creating a function using the `def` statement, you can specify the return value with a `return` statement, which consists of the following:

- The `return` keyword
- The value or expression that the function should return

In the case of an expression, the return value is whatever this expression evaluates to. For example, the following program defines a function that returns a different string depending on the number it is passed as an argument. Enter this code into the file editor and save it as *magic8Ball.py*:

```
❶ import random

❷ def get_answer(answer_number):
      # Returns a fortune answer based on what int answer_number is, 1 to 9
    ❸ if answer_number == 1:
          return 'It is certain'
      elif answer_number == 2:
          return 'It is decidedly so'
      elif answer_number == 3:
          return 'Yes'
      elif answer_number == 4:
          return 'Reply hazy try again'
      elif answer_number == 5:
          return 'Ask again later'
      elif answer_number == 6:
          return 'Concentrate and ask again'
      elif answer_number == 7:
          return 'My reply is no'
      elif answer_number == 8:
          return 'Outlook not so good'
      elif answer_number == 9:
          return 'Very doubtful'

  print('Ask a yes or no question:')
  input('>')
  ❹ r = random.randint(1, 9)
  ❺ fortune = get_answer(r)
  ❻ print(fortune)
```

When the program starts, Python first imports the random module ❶. Then comes the definition of the get_answer() function ❷. Because the function isn't being called, the code inside it is not run. Next, the program calls the random.randint() function with two arguments: 1 and 9 ❹. This function evaluates a random integer between 1 and 9 (including 1 and 9 themselves), then stores it in a variable named r.

Now the program calls the get_answer() function with r as the argument ❺. The program execution moves to the top of that function ❸, storing the value r in a parameter named answer_number. Then, depending on the value in answer_number, the function returns one of many possible string values. The execution returns to the line at the bottom of the program that originally called get_answer() ❺ and assigns the returned string to a variable named fortune, which then gets passed to a print() call ❻ and printed to the screen.

Note that because you can pass return values as arguments to other function calls, you could shorten these three lines

```
r = random.randint(1, 9)
fortune = get_answer(r)
print(fortune)
```

to this single equivalent line:

```
print(get_answer(random.randint(1, 9)))
```

Remember that expressions consist of values and operators; you can use a function call in an expression because the call evaluates to its return value.

The None Value

In Python, a value called None represents the absence of a value. The None value is the only value of the NoneType data type. (Other programming

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

languages might call this value null, nil, or undefined.) Just like the Boolean True and False values, you must always write None with a capital *N*.

This value-without-a-value can be helpful when you need to store something that shouldn't be confused for a real value in a variable. One place where None is used is as the return value of print(). The print() function displays text on the screen, and doesn't need to return anything in the same way len() or input() does. But since all function calls need to evaluate to a return value, print() returns None. To see this in action, enter the following into the interactive shell:

```
>>> spam = print('Hello!')
Hello!
>>> None == spam
True
```

Behind the scenes, Python adds return None to the end of any function definition with no return statement. This behavior resembles the way in which a while or for loop implicitly ends with a continue statement. Functions also return None if you use a return statement without a value (that is, just the return keyword by itself).

Named Parameters

Python identifies most arguments by their position in the function call. For example, random.randint(1, 10) is different from random.randint(10, 1). The first call returns a random integer between 1 and 10 because the first argument determines the low end of the range and the next argument determines its high end, while the second function call causes an error.

On the other hand, Python identifies *named parameters* by the name placed before them in the function call. You'll also hear named parameters called keyword parameters or keyword arguments, though they have nothing to do with Python keywords. Programmers often use named parameters to provide optional arguments. For example, the print() function uses the optional parameters end and sep to specify separator characters to print at the end of its arguments and between its arguments, respectively. If you ran a program without these arguments

```
print('Hello')
print('World')
```

the output would look like this:

```
Hello
World
```

The two strings appear on separate lines because the print() function automatically adds a newline character to the end of the string it is passed.

However, you can set the end named parameter to change the newline character to a different string. For example, if the code were this

```
print('Hello', end='')
print('World')
```

the output would look like this:

```
HelloWorld
```

The output appears on a single line because Python no longer prints a newline after 'Hello'. Instead, it prints a blank string. This is useful if you need to disable the newline that gets added to the end of every print() function call. Say you wanted to print the heads-or-tails results of a series of coin flips. Printing the output on a single line makes the output prettier, as in this *coinflip.py* program:

```
import random
for i in range(100):  # Perform 100 coin flips.
    if random.randint(0, 1) == 0:
        print('H', end=' ')
    else:
        print('T', end=' ')
print()  # Print one newline at the end.
```

This program displays the H and T results on one compact line, instead of spreading them out with one H or T result per line:

```
T H T T T H H T T T T H H H H T H H T T T T T H T T T T T H T T T T T H T H T
H H H T T H T T T T H T H H H T H H T H T T T T T H T T H T T T T H T H H H T
T T T H T T T T H H H T H T H H H H T H H T
```

Similarly, when you pass multiple string values to print(), the function automatically separates them with a single space. To see this behavior, enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice')
cats dogs mice
```

You could replace the default separating string by passing the sep named parameter a different string. Enter the following into the interactive shell:

```
>>> print('cats', 'dogs', 'mice', sep=',')
cats,dogs,mice
```

You can add named parameters to the functions you write as well, but first, you'll have to learn about the list and dictionary data types in Chapters 6 and 7. For now, just know that some functions have optional named parameters you can specify when calling the function.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The Call Stack

Imagine that you had a meandering conversation with someone. You talked about your friend Alice, which then reminded you of a story about your co-worker Bob, but first you had to explain something about your cousin Carol. You finished your story about Carol and went back to talking about Bob, and when you finished your story about Bob, you went back to talking about Alice. But then you were reminded about your brother David, so you told a story about him, and then you got back to finishing your original story about Alice. Your conversation followed a *stack*-like structure, like in Figure 4-1. In a stack, items get added or removed from the top only, and the current topic is always at the top of the stack.

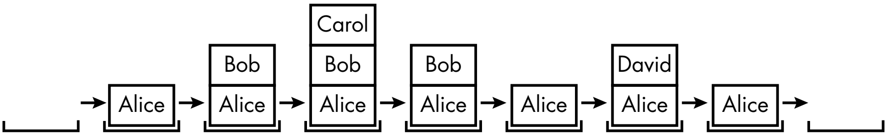


Figure 4-1: Your meandering conversation stack

Like your meandering conversation, calling a function doesn't send the execution on a one-way trip to the top of a function. Python will remember which line of code called the function so that the execution can return there when it encounters a `return` statement. If that original function called other functions, the execution would return to *those* function calls first, before returning from the original function call. The function call at the top of the stack is the execution's current location.

Open a file editor window and enter the following code, saving it as *abcdCallStack.py*:

```
def a():
    print('a() starts')
  ❶ b()
  ❷ d()
    print('a() returns')

def b():
    print('b() starts')
  ❸ c()
    print('b() returns')

def c():
  ❹ print('c() starts')
    print('c() returns')

def d():
    print('d() starts')
    print('d() returns')

❺ a()
```

If you run this program, the output will look like this:

```
a() starts
b() starts
c() starts
c() returns
b() returns
d() starts
d() returns
a() returns
```

When `a()` is called ❺, it calls `b()` ❶, which in turn calls `c()` ❸. The `c()` function doesn't call anything; it just displays `c() starts` ❹ and `c() returns` before returning to the line in `b()` that called it ❸. Once the execution returns to the code in `b()` that called `c()`, it returns to the line in `a()` that called `b()` ❶. The execution continues to the next line in the `a()` function ❷, which is a call to `d()`. Like the `c()` function, the `d()` function also doesn't call anything. It just displays `d() starts` and `d() returns` before returning to the line in `a()` that called it. Because `d()` contains no other code, the execution returns to the line in `a()` that called `d()` ❷. The last line in `a()` displays `a() returns` before returning to the original `a()` call at the end of the program ❺.

The *call stack* is how Python remembers where to return the execution after each function call. The call stack isn't stored in a variable in your program; rather, it's a section of your computer's memory that Python handles automatically behind the scenes. When your program calls a function, Python creates a *frame object* on the top of the call stack. Frame objects store the line number of the original function call so that Python can remember where to return. If the program makes another function call, Python adds another frame object above the other one on the call stack.

When a function call returns, Python removes a frame object from the top of the stack and moves the execution to the line number stored in it. Note that frame objects always get added and removed from the top of the stack, and not from any other place. Figure 4-2 illustrates the state of the call stack in *abcdCallStack.py* as each function is called and returns.

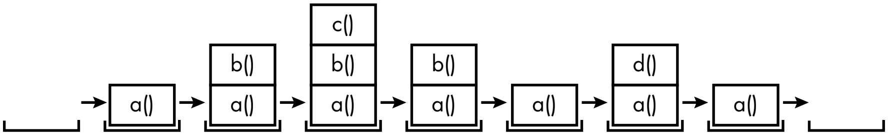


Figure 4-2: The frame objects of the call stack as abcdCallStack.py *calls and returns from functions*

The top of the call stack is the currently executing function. When the call stack is empty, the execution is on a line outside all functions.

The call stack is a technical detail that you don't strictly need to know about to write programs. It's enough to understand that function calls return to the line number they were called from. However, understanding call stacks makes it easier to understand local and global scopes, described in the next section.

Local and Global Scopes

Only code within a called function can access the parameters and variables assigned in that function. These variables are said to exist in that function's *local scope*. By contrast, code anywhere in a program can access variables that are assigned outside all functions. These variables are said to exist in the *global scope*. A variable that exists in a local scope is called a *local variable*, while a variable that exists in a global scope is called a *global variable*. A variable must be one or the other; it cannot be both local and global.

Think of a *scope* as a container for variables. There is only one global scope, created when your program begins. When your program terminates, it destroys the global scope, and all of its variables get forgotten. A new local scope gets created whenever a program calls a function. Any variables assigned in the function exist within the function's local scope. When the function returns, the local scope gets destroyed, along with these variables.

Python uses scoping because it enables a function to modify its variables, yet interact with the rest of the program through its parameters and its return value only. This narrows down the number of lines of code that might be causing a bug. If your program contained nothing but global variables, and contained a bug caused by a variable set to a bad value, you might struggle to track down the location of this bad value. It could have been set from anywhere in the program, which could be hundreds or thousands of lines long! But if the bug occurred in a local variable, you can restrict your search to a single function.

For this reason, while using global variables in small programs is fine, it's a bad habit to rely on global variables as your programs get larger and larger.

Scope Rules

When working with local and global variables, keep the following rules in mind:

- Code that is in the global scope, outside all functions, can't use local variables.
- Code that is in one function's local scope can't use variables in any other local scope.
- Code in a local scope can access global variables.
- You can use the same name for different variables if they are in different scopes. That is, there can be a local variable named `spam` and a global variable also named `spam`.

Let's review these rules with examples.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Code That Is in the Global Scope Can't Use Local Variables

Consider the following code, which will cause an error when you run it:

```
def spam():
  ❶ eggs = 'sss'
spam()
print(eggs)
```

The program's output will look like this:

```
Traceback (most recent call last):
  File "C:/test1.py", line 4, in <module>
    print(eggs)
NameError: name 'eggs' is not defined
```

The error happens because the eggs variable exists only in the local scope created when spam() is called ❶. Once the program execution returns from spam(), that local scope gets destroyed, and there is no longer a variable named eggs. So when your program tries to run print(eggs), Python gives you an error saying that eggs is not defined. This makes sense if you think about it; when the program execution is in the global scope, no local scopes exist, so there can't be any local variables. This is why you can only reference global variables in the global scope.

Code That Is in a Local Scope Can't Use Variables in Other Local Scopes

Python creates a new local scope whenever a program calls a function, even when the function is called from another function. Consider this program:

```
def spam():
    eggs = 'SPAMSPAM'
  ❶ bacon()
  ❷ print(eggs)  # prints 'SPAMSPAM'

def bacon():
    ham = 'hamham'
    eggs = 'BACONBACON'

❸ spam()
```

When the program starts, it calls the spam() function ❸, creating a local scope. The spam() function sets the local variable eggs to 'SPAMSPAM', then calls the bacon() function ❶, creating a second local scope. Multiple local scopes can exist at the same time. In this new local scope, the local variable ham gets set to 'hamham', and a local variable eggs (which differs from the one in spam()'s local scope) gets created and set to 'BACONBACON'. At this point, the program has two local variables named eggs that exist simultaneously: one that is local to spam() and one that is local to bacon().

When bacon() returns, Python destroys the local scope for that call, including its eggs variable. The program execution continues in the spam()

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

function, printing the value of eggs ❷. Because the local scope for the call to spam() still exists, the only eggs variable is the spam() function's eggs variable, which was set to 'SPAMSPAM'. This is what the program prints.

Code That Is in a Local Scope Can Use Global Variables

So far, I've demonstrated that code in the global scope can't access variables in a local scope; nor can code in a different local scope. Now consider the following program:

```
def spam():
    print(eggs)  # prints 'GLOBALGLOBAL'
eggs = 'GLOBALGLOBAL'
spam()
print(eggs)
```

Because the spam() function has no parameter named eggs, nor any code that assigns eggs a value, Python considers the function's use of eggs a reference to the global variable eggs. This is why the program prints 'GLOBALGLOBAL' when it's run.

Local and Global Variables Can Have the Same Name

Technically, it's perfectly acceptable to use the same variable name for a global variable and local variables in different scopes. But, to simplify your life, avoid doing this. To see what could happen, enter the following code into the file editor and save it as *localGlobalSameName.py*:

```
def spam():
  ❶ eggs = 'spam local'
    print(eggs)  # Prints 'spam local'

def bacon():
  ❷ eggs = 'bacon local'
    print(eggs)  # Prints 'bacon local'
    spam()
    print(eggs)  # Prints 'bacon local'

❸ eggs = 'global'
bacon()
print(eggs)  # Prints 'global'
```

When you run this program, it outputs the following:

```
bacon local
spam local
bacon local
global
```

This program actually contains three different variables, but confusingly, they're all named eggs. The variables are as follows:

- A variable named eggs that exists in a local scope when spam() is called ❶
- A variable named eggs that exists in a local scope when bacon() is called ❷
- A variable named eggs that exists in the global scope ❸

Because these three separate variables all have the same name, it can be hard to keep track of the one in use at any given time. Instead, give all variables unique names, even when they appear in different scopes.

The global Statement

If you need to modify a global variable from within a function, use the global statement. Including a line such as global eggs at the top of a function tells Python, "In this function, eggs refers to the global variable, so don't create a local variable with this name." For example, enter the following code into the file editor and save it as *globalStatement.py*:

```
def spam():
  ❶ global eggs
  ❷ eggs = 'spam'

eggs = 'global'
spam()
print(eggs)  # Prints 'spam'
```

When you run this program, the final print() call will output this:

```
spam
```

Because eggs is declared global at the top of spam() ❶, setting eggs to 'spam' ❷ changes the value of the globally scoped eggs. No local eggs variable is ever created.

Scope Identification

Use these four rules to tell whether a variable belongs to a local scope or the global scope:

1. A variable in the global scope (that is, outside all functions) is always a global variable.
2. A variable in a function with a global statement is always a global variable in that function.
3. Otherwise, if a function uses a variable in an assignment statement, it is a local variable.
4. However, if the function uses a variable but never in an assignment statement, it is a global variable.

To get a better feel for these rules, here's an example program. Enter the following code into the file editor and save it as *sameNameLocalGlobal.py*:

```
def spam():
  ❶ global eggs
    eggs = 'spam'  # This is the global variable.

def bacon():
  ❷ eggs = 'bacon'  # This is a local variable.

def ham():
  ❸ print(eggs)  # This is the global variable.

eggs = 'global'  # This is the global variable.
spam()
print(eggs)
```

In the `spam()` function, `eggs` refers to the global `eggs` variable because the function includes a `global` statement for it ❶. In `bacon()`, `eggs` is a local variable because the function includes an assignment statement for it ❷. In `ham()` ❸, `eggs` is the global variable because the function contains no assignment statement or `global` statement for the variable. If you run *sameNameLocalGlobal.py*, the output will look like this:

```
spam
```

If you try to use a local variable in a function before you assign a value to it, as in the following program, Python will give you an error. To see this, enter the following into the file editor and save it as *sameNameError.py*:

```
def spam():
    print(eggs)  # ERROR!
  ❶ eggs = 'spam local'

❷ eggs = 'global'
spam()
```

If you run the previous program, it produces an error message:

```
Traceback (most recent call last):
  File "C:/sameNameError.py", line 6, in <module>
    spam()
  File "C:/sameNameError.py", line 2, in spam
    print(eggs)  # ERROR!
UnboundLocalError: local variable 'eggs' referenced before assignment
```

This error happens because Python sees that there is an assignment statement for `eggs` in the `spam()` function ❶ and, therefore, considers any mention of an `eggs` variable in `spam()` to be a local variable. But because `print(eggs)` is executed before `eggs` is assigned anything, the local variable `eggs` doesn't exist. Python won't fall back to using the global `eggs` variable ❷.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The name of the error, UnboundLocalError, can be somewhat confusing. In Python, *binding* is another way of saying *assigning*, so this error indicates that the program used a local variable before it was assigned a value.

FUNCTIONS AS "BLACK BOXES"

Often, all you need to know about a function are its inputs (the parameters) and its output value; you don't always have to burden yourself with how the function's code actually works. When you think about functions in this high-level way, it's common to say that you're treating a function as a "black box."

This idea is fundamental to modern programming. Later chapters in this book will show you several modules with functions that were written by other people. While you can take a peek at the source code if you're curious, you don't need to know how these functions work in order to use them. And because writing functions without global variables is encouraged, you usually don't have to worry about the function's code interacting with the rest of your program.

Exception Handling

Right now, getting an error, or *exception*, in your Python program means the entire program will crash. You don't want this to happen in real-world programs. Instead, you want the program to detect errors, handle them, and then continue to run.

For example, consider the following program, which has a divide-by-zero error. Open a file editor window and enter the following code, saving it as *zeroDivide.py*:

```
def spam(divide_by):
    return 42 / divide_by

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

We've defined a function called spam, given it a parameter, and then printed the value of that function with various parameters to see what happens. This is the output you get when you run the previous code:

```
21.0
3.5
Traceback (most recent call last):
  File "C:/zeroDivide.py", line 6, in <module>
    print(spam(0))
  File "C:/zeroDivide.py", line 2, in spam
    return 42 / divide_by
ZeroDivisionError: division by zero
```

A `ZeroDivisionError` happens whenever you try to divide a number by zero. From the line number given in the error message, you know that the `return` statement in `spam()` is causing an error.

Most of the time, exceptions indicate a bug in your code that you need to fix. But sometimes exceptions can be expected and recovered from. For example, in Chapter 10 you'll learn how to read text from files. If you specify a filename for a file that doesn't exist, Python raises a `FileNotFoundError` exception. You might want to handle this exception by asking the user to enter the filename again rather than having this unhandled exception immediately crash your program.

Errors can be handled with `try` and `except` statements. The code that could potentially have an error is put in a `try` clause. The program execution moves to the start of a following `except` clause if an error happens.

You can put the previous divide-by-zero code in a `try` clause and have an `except` clause contain code to handle what happens when this error occurs:

```
def spam(divideBy):
    try:
        # Any code in this block that causes ZeroDivisionError won't crash the program:
        return 42 / divideBy
    except ZeroDivisionError:
        # If ZeroDivisionError happened, the code in this block runs:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

When code in a `try` clause causes an error, the program execution immediately moves to the code in the `except` clause. After running that code, the execution continues as normal. If the program doesn't raise an exception in the `try` clause, the program skips the code in the `except` clause. The output of the previous program is as follows:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

Note that any errors that occur in function calls in a `try` block will also be caught. Consider the following program, which instead has the `spam()` calls in the `try` block:

```
def spam(divideBy):
    return 42 / divideBy
```

```
try:
    print(spam(2))
    print(spam(12))
    print(spam(0))
    print(spam(1))
except ZeroDivisionError:
    print('Error: Invalid argument.')
```

When this program is run, the output looks like this:

```
21.0
3.5
Error: Invalid argument.
```

The reason print(spam(1)) is never executed is because once the execution jumps to the code in the except clause, it does not return to the try clause. Instead, it just continues moving down the program as normal.

A Short Program: Zigzag

Let's use the programming concepts you've learned so far to create a small animation program. This program will create a back-and-forth zigzag pattern until the user stops it by pressing the Mu editor's Stop button or by pressing CTRL-C. When you run this program, the output will look something like this:

```
    ********
   ********
  ********
 ********
********
 ********
  ********
   ********
    ********
```

Enter the following source code into the file editor, and save the file as *zigzag.py*:

```
import time, sys
indent = 0  # How many spaces to indent
indentIncreasing = True  # Whether the indentation is increasing or not

try:
    while True:  # The main program loop
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1) # Pause for 1/10th of a second.

        if indentIncreasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                # Change direction:
                indentIncreasing = False
        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                # Change direction:
                indentIncreasing = True
except KeyboardInterrupt:
    sys.exit()
```

Let's look at this code line by line, starting at the top:

```
import time, sys
indent = 0 # How many spaces to indent.
indentIncreasing = True  # Whether the indentation is increasing or not
```

First, we'll import the `time` and `sys` modules. Our program uses two variables. The `indent` variable keeps track of how many spaces of indentation occur before the band of eight asterisks, and the `indentIncreasing` variable contains a Boolean value to determine whether the amount of indentation is increasing or decreasing:

```
try:
    while True:  # The main program loop
        print(' ' * indent, end='')
        print('********')
        time.sleep(0.1) # Pause for 1/10 of a second.
```

Next, we place the rest of the program inside a `try` statement. When the user presses CTRL-C while a Python program is running, Python raises the `KeyboardInterrupt` exception. If there is no `try`-`except` statement to catch this exception, the program crashes with an ugly error message. However, we want our program to cleanly handle the `KeyboardInterrupt` exception by calling `sys.exit()`. (You can find the code that accomplishes this in the `except` statement at the end of the program.)

The `while True:` infinite loop will repeat the instructions in the program forever. This involves using `' ' * indent` to print the correct number of spaces for the indentation. We don't want to automatically print a newline after these spaces, so we also pass `end=''` to the first `print()` call. A second `print()` call prints the band of asterisks. We haven't discussed the `time.sleep()` function yet; suffice it to say that it introduces a one-tenth-of-a-second pause in our program:

```
        if indentIncreasing:
            # Increase the number of spaces:
            indent = indent + 1
            if indent == 20:
                indentIncreasing = False  # Change direction
```

Next, we want to adjust the amount of indentation used the next time we print asterisks. If `indentIncreasing` is `True`, we'll add `1` to `indent`, but once indent reaches `20`, we'll decrease the indentation:

```
        else:
            # Decrease the number of spaces:
            indent = indent - 1
            if indent == 0:
                indentIncreasing = True  # Change direction
```

If `indentIncreasing` is `False`, we'll want to subtract one from `indent`. Once indent reaches `0`, we'll want the indentation to increase once again. Either way, the program execution will jump back to the start of the main program loop to print the asterisks again.

If the user presses CTRL-C at any point that the program execution is in the `try` block, this `except` statement raises and handles the `KeyboardInterrupt` exception:

```
except KeyboardInterrupt:
    sys.exit()
```

The program execution moves inside the `except` block, which runs `sys.exit()` and quits the program. This way, even though the main program loop is infinite, the user has a way to shut down the program.

A Short Program: Spike

Let's create another scrolling animation program. This program uses string replication and nested loops to draw spikes. Open a new file in your code editor, enter the following code, and save it as *spike.py*:

```
import time, sys

try:
    while True:  # The main program loop
        # Draw lines with increasing length:
        for i in range(1, 9):
            print('-' * (i * i))
            time.sleep(0.1)

        # Draw lines with decreasing length:
        for i in range(7, 1, -1):
            print('-' * (i * i))
            time.sleep(0.1)
except KeyboardInterrupt:
    sys.exit()
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

When you run this program, it repeatedly draws the following spike:

```
-
----
---------
----------------
-------------------------
------------------------------------
-------------------------------------------------
----------------------------------------------------------------
-------------------------------------------------
------------------------------------
-------------------------
----------------
---------
----
```

Like the previous zigzag program, the spike program needs to call the `time.sleep()` and `sys.exit()` functions. The first line imports the `time` and `sys` modules. A `try` block will catch the `KeyboardInterrupt` exception raised when the user presses CTRL-C, and an infinite loop continues drawing spikes forever.

The first `for` loop draws spikes of increasing sizes:

```
        # Draw lines with increasing length:
        for i in range(1, 9):
            print('-' * (i * i))
            time.sleep(0.1)
```

Because the `i` variable is set to `1`, then `2`, then `3`, and so on, up to but not including `9`, the following `print()` call replicates the `'-'` strings by `1 * 1` (that is, `1`), then `2 * 2` (that is, `4`), then `3 * 3` (that is, `9`), and so on. This code is what creates the strings that are 1, 4, 9, 16, 25, 36, 49, and then 64 dashes long. By having exponentially longer strings, we create the top half of the spike.

To draw the bottom half of the spike, we need another `for` loop that causes `i` to start at `7` and then decrease down to `1`, not including `1`:

```
        # Draw lines with decreasing length:
        for i in range(7, 1, -1):
            print('-' * (i * i))
            time.sleep(0.1)
```

You can modify the `9` and the `7` values in the two `for` loops if you want to change how wide the spike becomes. The rest of the code will continue to work just fine with these new values.

Summary

Functions are the primary way to compartmentalize your code into logical groups. Since the variables in functions exist in their own local scopes, the code in one function cannot directly affect the values of variables in other functions. This limits the sections of code able to change the values of your variables, which can be helpful when it comes to debugging.

Functions are a great tool to help you organize your code. You can think of them as black boxes: they have inputs in the form of parameters and outputs in the form of return values, and the code in them doesn't affect variables in other functions.

In previous chapters, a single error could cause your programs to crash. In this chapter, you learned about `try` and `except` statements, which can run code when an error has been detected. This can make your programs more resilient to common error cases.

Practice Questions

1. Why are functions advantageous to have in your programs?
2. When does the code in a function execute: when the function is defined or when the function is called?
3. What statement creates a function?
4. What is the difference between a function and a function call?
5. How many global scopes are there in a Python program? How many local scopes are there?
6. What happens to variables in a local scope when the function call returns?
7. What is a return value? Can a return value be part of an expression?
8. If a function does not have a `return` statement, what is the return value of a call to that function?
9. How can you force a variable in a function to refer to the global variable?
10. What is the data type of `None`?
11. What does the `import areallyourpetsnamederic` statement do?
12. If you had a function named `bacon()` in a module named `spam`, how would you call it after importing `spam`?
13. How can you prevent a program from crashing when it gets an error?
14. What goes in the `try` clause? What goes in the `except` clause?
15. Write the following program in a file named *notrandomdice.py* and run it. Why does each function call return the same number?

```
import random
random_number = random.randint(1, 6)

def get_random_dice_roll():
 # Returns a random integer from 1 to 6.
 return random_number
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
print(get_random_dice_roll())
print(get_random_dice_roll())
print(get_random_dice_roll())
print(get_random_dice_roll())
```

Practice Programs

For practice, write programs to do the following tasks.

The Collatz Sequence

Write a function named `collatz()` that has one parameter named `number`. If `number` is even, then `collatz()` should print `number // 2` and return this value. If `number` is odd, then `collatz()` should print and return `3 * number + 1`.

Then, write a program that lets the user enter an integer and that keeps calling `collatz()` on that number until the function returns the value `1`. (Amazingly enough, this sequence actually works for any integer; sooner or later, using this sequence, you'll arrive at 1! Even mathematicians aren't sure why. Your program is exploring what's called the *Collatz sequence*, sometimes called "the simplest impossible math problem.")

Remember to convert the return value from `input()` to an integer with the `int()` function; otherwise, it will be a string value. To make the output more compact, the `print()` calls that print the numbers should have a `sep=' '` named parameter to print all values on one line.

The output of this program could look something like this:

```
Enter number:
3
3 10 5 16 8 4 2 1
```

Hint: An integer `number` is even if `number % 2 == 0`, and it's odd if `number % 2 == 1`.

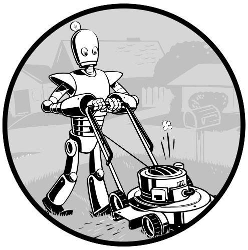
Input Validation

Add `try` and `except` statements to the previous project to detect whether the user entered a non-integer string. Normally, the `int()` function will raise a `ValueError` error if it is passed a non-integer string, as in `int('puppy')`. In the `except` clause, print a message to the user saying they must enter an integer.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

5

DEBUGGING



Now that you know enough to write basic programs, you may start finding not-so-simple bugs in them. This chapter covers some tools and techniques for finding the root cause of bugs in your program to help you fix them more quickly and with less effort. To paraphrase an old joke among programmers, writing code accounts for 90 percent of programming. Debugging code accounts for the other 90 percent.

Your computer will do only what you tell it to do; it won't read your mind and do what you *intended* it to do. Even professional programmers create bugs all the time, so don't feel discouraged if your program has a problem.

Fortunately, a few tools and techniques can identify exactly what your code is doing and where it's going wrong. You'll use the *debugger,* a feature of Mu that executes a program one instruction at a time, giving you a chance to inspect the values in variables while your code runs, and track

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

how the values change over the course of your program. This process is much slower than running the program at full speed, but it allows you to see the actual values in a program while it runs, rather than having to deduce what the values might be from the source code.

You'll also make your programs raise custom exceptions to indicate errors, and you'll learn about logging and assertions, two features that can help you detect bugs early. In general, the earlier you catch bugs, the easier they will be to fix.

Raising Exceptions

Python raises an exception whenever it tries to execute invalid code. In Chapter 4, you handled Python's exceptions with `try` and `except` statements so that your program could recover from exceptions you anticipated. But you can also raise your own exceptions in your code. Raising an exception is a way of saying, "Stop running this code, and move the program execution to the `except` statement."

We raise exceptions with a `raise` statement, which consists of the following:

- The `raise` keyword
- A call to the `Exception()` function
- A string with a helpful error message passed to the `Exception()` function

For example, enter the following into the interactive shell:

```
>>> raise Exception('This is the error message.')
Traceback (most recent call last):
  File "<pyshell#191>", line 1, in <module>
    raise Exception('This is the error message.')
Exception: This is the error message.
```

If no `try` and `except` statements cover the `raise` statement that raised the exception, the program simply crashes and displays the exception's error message.

Often, it's the code that calls the function containing a `raise` statement, rather than the function itself, that knows how to handle an exception. That means you'll commonly see a `raise` statement inside a function, and the `try` and `except` statements in the code calling the function. For example, open a new file editor tab, enter the following code, and save the program as *boxPrint.py*:

```
def box_print(symbol, width, height):
    if len(symbol) != 1:
      ❶ raise Exception('Symbol must be a single character string.')
    if width <= 2:
      ❷ raise Exception('Width must be greater than 2.')
    if height <= 2:
      ❸ raise Exception('Height must be greater than 2.')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    print(symbol * width)
    for i in range(height - 2):
        print(symbol + (' ' * (width - 2)) + symbol)
    print(symbol * width)

try:
    box_print('*', 4, 4)
    box_print('O', 20, 5)
    box_print('x', 1, 3)
    box_print('ZZ', 3, 3)
❹ except Exception as err:
  ❺ print('An exception happened: ' + str(err))
try:
    box_print('ZZ', 3, 3)
except Exception as err:
    print('An exception happened: ' + str(err))
```

Here, we've defined a `box_print()` function that takes a character, a width, and a height, and uses the character to make a little picture of a box with that width and height. This box shape is printed to the screen.

Say we want the function to accept a single character only, and we expect the width and height to be greater than 2. We add `if` statements to raise exceptions if these requirements aren't satisfied. Later, when we call `box_print()` with various arguments, our `try`/`except` will handle invalid arguments.

This program uses the `except Exception as err` form of the `except` statement ❹. If an `Exception` object is returned from `box_print()` ❶ ❷ ❸, this `except` statement will store it in a variable named `err`. We can then convert the `Exception` object to a string by passing it to `str()` to produce a user-friendly error message ❺. When you run this *boxPrint.py*, the output will look like this:

```
****
*  *
*  *
****
OOOOOOOOOOOOOOOOOOOO
O                  O
O                  O
O                  O
OOOOOOOOOOOOOOOOOOOO
An exception happened: Width must be greater than 2.
An exception happened: Symbol must be a single character string.
```

Using the `try` and `except` statements, you can handle errors gracefully, rather than letting the entire program crash.

Assertions

An *assertion* is a sanity check to make sure your code isn't doing something obviously wrong. We perform these sanity checks with `assert` statements. If the sanity check fails, the code raises an AssertionError exception. An assert statement consists of the following:

- The assert keyword
- A condition (that is, an expression that evaluates to True or False)
- A comma
- A string to display when the condition is False

In plain English, an assert statement says, "I assert that the condition holds true, and if not, there is a bug somewhere, so immediately stop the program." For example, enter the following into the interactive shell:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.sort()
>>> ages
[15, 17, 22, 26, 47, 54, 57, 73, 80, 92]
>>> assert ages[0] <= ages[-1]  # Assert that the first age is <= the last age.
```

The assert statement here asserts that the first item in ages should be less than or equal to the last one. This is a sanity check; if the code in sort() is bug-free and did its job, then the assertion would be true. Because the ages[0] <= ages[-1] expression evaluates to True, the assert statement does nothing.

However, let's pretend we had a bug in our code. Say we accidentally called the reverse() list method instead of the sort() list method. When we enter the following in the interactive shell, the assert statement raises an AssertionError:

```
>>> ages = [26, 57, 92, 54, 22, 15, 17, 80, 47, 73]
>>> ages.reverse()
>>> ages
[73, 47, 80, 17, 15, 22, 54, 92, 57, 26]
>>> assert ages[0] <= ages[-1]  # Assert that the first age is <= the last age.
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AssertionError
```

Unlike exceptions, your code should *not* handle assert statements with try and except; if an assert fails, your program *should* crash. By "failing fast" like this, you shorten the time between the original cause of the bug and the moment you first notice it, reducing the amount of code you'll have to check before finding the bug's cause.

Assertions are for programmer errors, not user errors. Assertions should fail only while the program is under development; a user should never see an assertion error in a finished program. For errors that your program can encounter as a normal part of its operation (such as a file not being found or the user entering invalid data), raise an exception instead of detecting it with an assert statement.

Logging

If you've ever put a `print()` function in your code to output some variable's value while your program is running, you've used a form of *logging* to debug your code. Logging is a great way to understand what's happening in your program and in what order it's happening. Python's `logging` module makes it easy to create a record of custom messages that you write. These log messages will describe when the program execution has reached the logging function call and will list any variables you've specified at that point in time, providing a trail of breadcrumbs that can help you figure out when things started to go wrong. On the other hand, a missing log message indicates a part of the code was skipped and never executed.

The logging Module

To enable the `logging` module to display log messages on your screen as your program runs, copy the following to the top of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -  %(levelname)s -  %(message)s')
```

The `logging` module's `basicConfig()` function lets you specify what details you want to see and how you want those details displayed.

Say you wrote a function to calculate the *factorial* of a number. In mathematics, the factorial of 4 is 1 × 2 × 3 × 4, or 24. The factorial of 7 is 1 × 2 × 3 × 4 × 5 × 6 × 7, or 5,040. Open a new file editor tab and enter the following code. It has a bug in it, but you'll generate several log messages to help figure out what's going wrong. Save the program as *factorialLog.py*:

```
import logging
logging.basicConfig(level=logging.DEBUG, format='%(asctime)s -  %(levelname)s -  %(message)s')
logging.debug('Start of program')

def factorial(n):
    logging.debug('Start of factorial(' + str(n) + ')')
    total = 1
    for i in range(n + 1):
        total *= i
        logging.debug('i is ' + str(i) + ', total is ' + str(total))
    logging.debug('End of factorial(' + str(n) + ')')
    return total

print(factorial(5))
logging.debug('End of program')
```

We use the `logging.debug()` function to print log information. This `debug()` function calls `basicConfig()`, which prints a line of information in the format we specified in the function call, along with the messages we passed to `debug()`. The `print(factorial(5))` call is part of the original program, so the code displays the result even if logging messages are disabled.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The output of this program looks like this:

```
2035-05-23 16:20:12,664 - DEBUG - Start of program
2035-05-23 16:20:12,664 - DEBUG - Start of factorial(5)
2035-05-23 16:20:12,665 - DEBUG - i is 0, total is 0
2035-05-23 16:20:12,668 - DEBUG - i is 1, total is 0
2035-05-23 16:20:12,670 - DEBUG - i is 2, total is 0
2035-05-23 16:20:12,673 - DEBUG - i is 3, total is 0
2035-05-23 16:20:12,675 - DEBUG - i is 4, total is 0
2035-05-23 16:20:12,678 - DEBUG - i is 5, total is 0
2035-05-23 16:20:12,680 - DEBUG - End of factorial(5)
0
2035-05-23 16:20:12,684 - DEBUG - End of program
```

The `factorial()` function returns `0` as the factorial of `5`, which isn't right. The `for` loop should be multiplying the value in `total` by the numbers from `1` to `5`, but the log messages displayed by `logging.debug()` show that the `i` variable starts at `0` instead of `1`. Since zero times anything is zero, the rest of the iterations have the wrong value for `total`.

Change the `for i in range(n + 1):` line to `for i in range(`**`1,`**` n + 1):`, and run the program again. The output will look like this:

```
2035-05-23 17:13:40,650 - DEBUG - Start of program
2035-05-23 17:13:40,651 - DEBUG - Start of factorial(5)
2035-05-23 17:13:40,651 - DEBUG - i is 1, total is 1
2035-05-23 17:13:40,654 - DEBUG - i is 2, total is 2
2035-05-23 17:13:40,656 - DEBUG - i is 3, total is 6
2035-05-23 17:13:40,659 - DEBUG - i is 4, total is 24
2035-05-23 17:13:40,661 - DEBUG - i is 5, total is 120
2035-05-23 17:13:40,661 - DEBUG - End of factorial(5)
120
2035-05-23 17:13:40,666 - DEBUG - End of program
```

The `factorial(5)` call correctly returns `120`. The log messages showed what was going on inside the loop, which led straight to the bug.

You can see that the `logging.debug()` calls printed out not just the strings passed to them but also a timestamp and the word *DEBUG*.

Logfiles

Instead of displaying the log messages to the screen, you can write them to a text file. The `logging.basicConfig()` function takes a `filename` named parameter, like so:

```
import logging
logging.basicConfig(filename='myProgramLog.txt', level=logging.DEBUG,
format=' %(asctime)s -  %(levelname)s -  %(message)s')
```

This code will save the log messages to *myProgramLog.txt*.

While logging messages are helpful, they can clutter your screen and make it hard to read the program's output. Writing the logging messages to

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

a file will keep your screen clear and enable you to read the messages after running the program. You can open this text file in any text editor, such as Notepad or TextEdit.

A Poor Practice: Debugging with print()

Entering `import logging` and `logging.basicConfig(level=logging.DEBUG, format='%(asctime)s - %(levelname)s - %(message)s')` is somewhat unwieldy. You may want to use `print()` calls instead, but don't give in to this temptation! Once you're done debugging, you'll end up spending a lot of time removing `print()` calls from your code for each log message. You might even accidentally remove some `print()` calls that were used for non-log messages. The nice thing about log messages is that you're free to fill your program with as many as you like, and can always disable them later by adding a single `logging.disable(logging.CRITICAL)` call. Unlike `print()`, the `logging` module makes it easy to switch between showing and hiding log messages.

Log messages are intended for the programmer, not the user. The user won't care about the contents of some dictionary value you need to see to help with debugging; use a log message for something like that. For error messages that the user should see, like `File not found` or `Invalid input, please enter a number`, use a `print()` call. You don't want to deprive the user of helpful information they can use to solve their problem.

Logging Levels

Logging levels provide a way to categorize your log messages by importance so that you can filter less important messages while testing your programs. There are five logging levels, described in Table 5-1 from least to most important. Your program can log messages at each level using different logging functions.

Table 5-1: Logging Levels in Python

Level	Logging function	Description
DEBUG	`logging.debug()`	The lowest level, used for small details. Usually, you'll care about these messages only when diagnosing problems.
INFO	`logging.info()`	Used to record information about general events in your program or to confirm that it's working at various points.
WARNING	`logging.warning()`	Used to indicate a potential problem that doesn't prevent the program from working but might do so in the future.
ERROR	`logging.error()`	Used to record an error that caused the program to fail to do something.
CRITICAL	`logging.critical()`	The highest level, used to indicate a fatal error that has caused, or is about to cause, the program to stop running entirely.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Ultimately, it's up to you to decide which category your log message falls into. You can pass the log message to these functions as a string. Try this yourself by entering the following into the interactive shell:

```
>>> import logging
>>> logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.debug('Some minor code and debugging details.')
2035-05-18 19:04:26,901 - DEBUG - Some debugging details.
>>> logging.info('An event happened.')
2035-05-18 19:04:35,569 - INFO - The logging module is working.
>>> logging.warning('Something could go wrong.')
2035-05-18 19:04:56,843 - WARNING - An error message is about to be logged.
>>> logging.error('An error has occurred.')
2035-05-18 19:05:07,737 - ERROR - An error has occurred.
>>> logging.critical('The program is unable to recover and will now terminate!')
2035-05-18 19:05:45,794 - CRITICAL - The program is unable to recover!
```

The benefit of logging levels is that you can change the priority of the logging messages you want to see. Passing `logging.DEBUG` to the `basicConfig()` function's `level` named parameter will show messages from all the logging levels (DEBUG being the lowest level). But after developing your program some more, you may be interested only in errors. In that case, you can set `basicConfig()`'s `level` argument to `logging.ERROR`. This will show only ERROR and CRITICAL messages and will skip the DEBUG, INFO, and WARNING messages.

Disabled Logging

After you've debugged your program, you probably don't want all these log messages cluttering the screen. The `logging.disable()` function disables these so that you don't have to remove the logging calls by hand. Simply pass `logging.disable()` a logging level to suppress all log messages at that level or lower. To disable logging entirely, add `logging.disable(logging.CRITICAL)` to your program. For example, enter the following into the interactive shell:

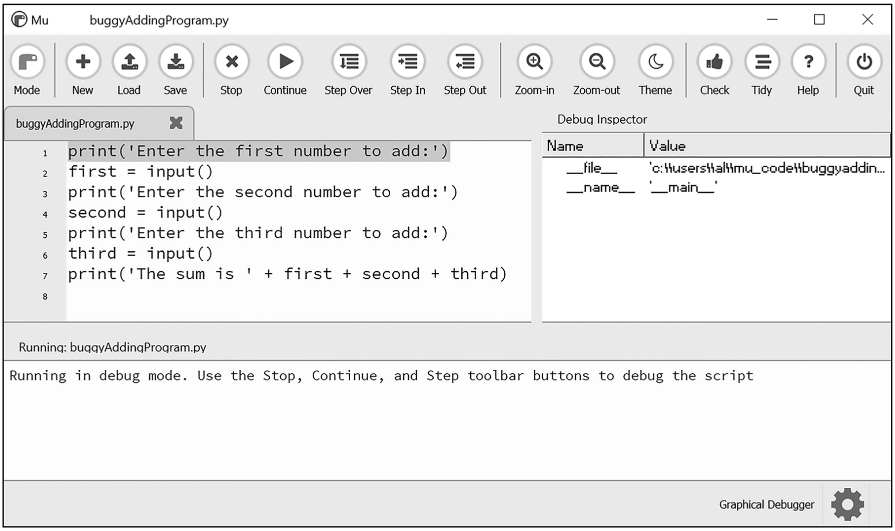
```
>>> import logging
>>> logging.basicConfig(level=logging.INFO, format=' %(asctime)s -
%(levelname)s -  %(message)s')
>>> logging.critical('Critical error! Critical error!')
2035-05-22 11:10:48,054 - CRITICAL - Critical error! Critical error!
>>> logging.disable(logging.CRITICAL)
>>> logging.critical('Critical error! Critical error!')
>>> logging.error('Error! Error!')
```

Because `logging.disable()` will disable all messages after it, you'll probably want to add it near the `import logging` line of code in your program. This way, you can easily find it to comment out or uncomment that call to enable or disable logging messages as needed.

Mu's Debugger

The *debugger* is a feature of the Mu editor, IDLE, and other editor software that allows you to execute your program one line at a time. The debugger will run a single line of code and then wait for you to tell it to continue. By running your program "under the debugger" like this, you can take as much time as you want to examine the values in the variables at any given point during the program's lifetime. This is a valuable tool for tracking down bugs.

To run a program under Mu's debugger, click the Debug button in the top row of buttons, next to the Run button. The Debug Inspector pane should open in the right side of the window. This pane lists the current value of variables in your program. In Figure 5-1, the debugger has paused the execution of the program just before it would have run the first line of code. You can see this line highlighted in the file editor.



```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

Name	Value
__file__	'c:\\users\\al\\mu_code\\buggyaddin…
__name__	'__main__'

```
Running in debug mode. Use the Stop, Continue, and Step toolbar buttons to debug the script
```

Figure 5-1: Mu running a program under the debugger

Debugging mode also adds the following new buttons to the top of the editor: Continue, Step Over, Step In, and Step Out. The usual Stop button is also available.

Clicking the Continue button will cause the program to execute normally until it terminates or reaches a *breakpoint.* (I'll describe breakpoints later in this chapter.) If you're done debugging and want the program to continue normally, click the Continue button.

Clicking the Step In button will cause the debugger to execute the next line of code and then pause again. If the next line of code is a function call, the debugger will *step into* that function, jumping to the function's first line of code.

Clicking the Step Over button will execute the next line of code, similar to the Step In button. However, if the next line of code is a function call,

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

the Step Over button will *step over*, or rush through, the code in the function. The function's code will execute at full speed, and the debugger will pause as soon as the function call returns. For example, if the next line of code calls a `spam()` function but you don't really care about code inside this function, you can click Step Over to execute the code in the function at normal speed, and then pause when the function returns. For this reason, using the Step Over button is more common than using the Step In button.

Clicking the Step Out button will cause the debugger to execute lines of code at full speed until it returns from the current function. If you've stepped into a function call with the Step In button and now simply want to keep executing instructions until you leave it, click the Step Out button to *step out* of the current function call.

If you want to stop debugging entirely and not bother to continue executing the rest of the program, click the Stop button. The Stop button will immediately terminate the program.

Debugging an Addition Program

To practice using the Mu debugger, open a new file editor tab and enter the following code:

```
print('Enter the first number to add:')
first = input()
print('Enter the second number to add:')
second = input()
print('Enter the third number to add:')
third = input()
print('The sum is ' + first + second + third)
```

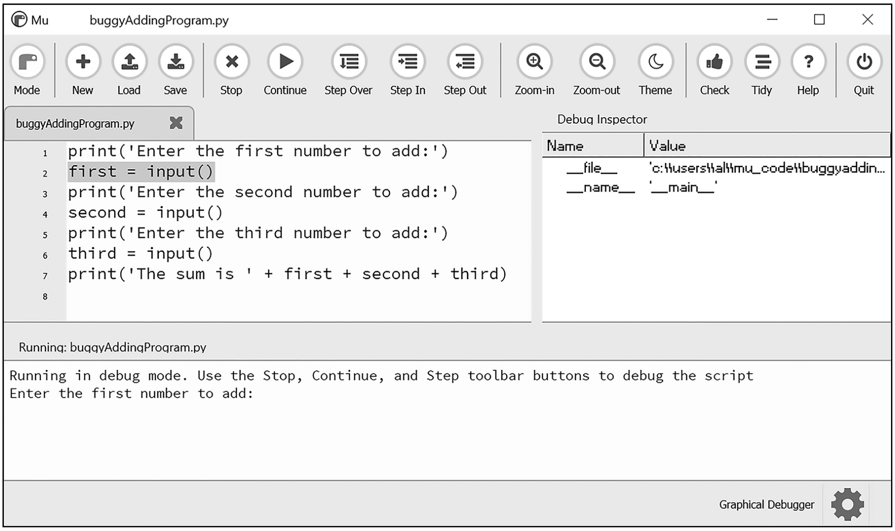
Save it as *buggyAddingProgram.py* and run it first without the debugger enabled. The program will output something like this:

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
The sum is 5342
```

The program hasn't crashed, but the sum is obviously wrong.

Run the program again, this time under the debugger. Click the **Debug** button, and the program should pause on line 1, which is the code it's about to execute.

Click the **Step Over** button once to execute the first `print()` call. You should use Step Over instead of Step In here because you don't want to step into the code for the `print()` function (although Mu should prevent the debugger from entering Python's built-in functions). The debugger moves on to line 2, and highlights line 2 in the file editor, as shown in Figure 5-2. This shows you where the program execution currently is.



```
1  print('Enter the first number to add:')
2  first = input()
3  print('Enter the second number to add:')
4  second = input()
5  print('Enter the third number to add:')
6  third = input()
7  print('The sum is ' + first + second + third)
8
```

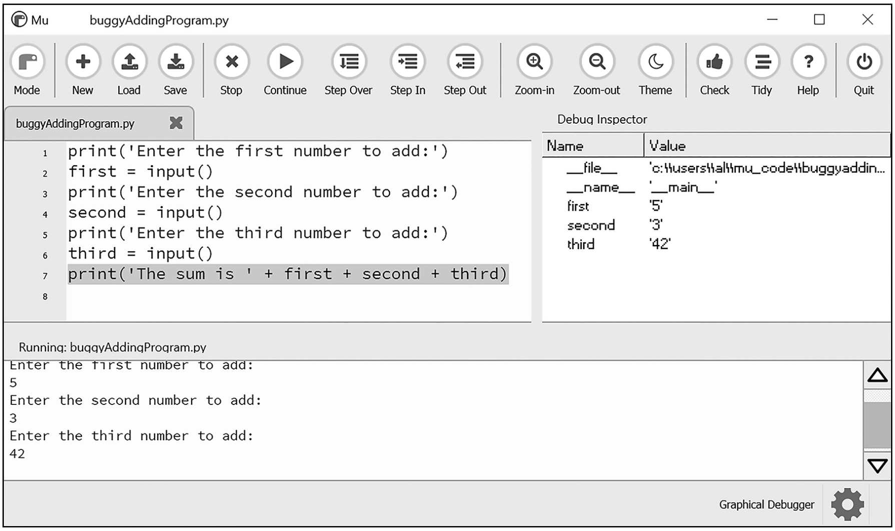
Name	Value
__file__	'c:\\users\\al\\mu_code\\buggyaddin...
__name__	'__main__'

```
Running in debug mode. Use the Stop, Continue, and Step toolbar buttons to debug the script
Enter the first number to add:
```

Figure 5-2: The Mu editor window after clicking Step Over

Click **Step Over** again to execute the `input()` function call. The highlighting will go away while Mu waits for you to type something for the `input()` call into the output pane. Enter **5** and press ENTER. The highlighting will return.

Keep clicking **Step Over**, and enter **3** and **42** as the next two numbers. When the debugger reaches line 7, the final `print()` call in the program, the Mu editor window should look like Figure 5-3.



```
1  print('Enter the first number to add:')
2  first = input()
3  print('Enter the second number to add:')
4  second = input()
5  print('Enter the third number to add:')
6  third = input()
7  print('The sum is ' + first + second + third)
8
```

Name	Value
__file__	'c:\\users\\al\\mu_code\\buggyaddin...
__name__	'__main__'
first	'5'
second	'3'
third	'42'

```
Enter the first number to add:
5
Enter the second number to add:
3
Enter the third number to add:
42
```

Figure 5-3: The Debug Inspector pane, located on the right side of the Mu editor window, shows that the variables are set to strings instead of integers, causing the bug.

In the Debug Inspector pane, you should see that the `first`, `second`, and `third` variables are set to string values `'5'`, `'3'`, and `'42'` instead of integer values `5`, `3`, and `42`. When the last line is executed, Python concatenates these strings instead of adding the numbers together, causing the bug.

Stepping through the program with the debugger is helpful but can also be slow. Often, you'll want the program to run normally until it reaches a certain line of code. You can configure the debugger to do this with breakpoints.

Setting Breakpoints

Setting a *breakpoint* on a specific line of code forces the debugger to pause whenever the program execution reaches that line. Open a new file editor tab and enter the following program, which simulates flipping a coin 1,000 times. Save it as *coinFlip.py*:

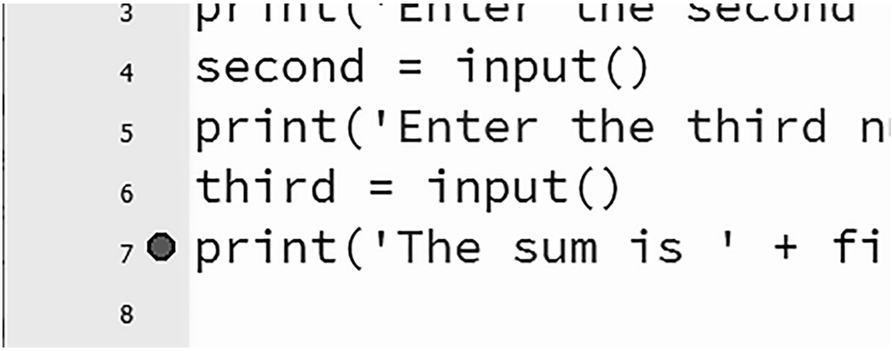
```
import random
heads = 0
for i in range(1, 1001):
  ❶ if random.randint(0, 1) == 1:
        heads = heads + 1
    if i == 500:
      ❷ print('Halfway done!')
print('Heads came up ' + str(heads) + ' times.')
```

The `random.randint(0, 1)` call ❶ will return `0` half of the time and `1` the other half of the time, simulating a 50/50 coin flip where `1` represents heads. When you run this program without the debugger, it quickly outputs something like the following:

```
Halfway done!
Heads came up 490 times.
```

If you ran this program under the debugger, you would have to click the Step Over button thousands of times before the program terminated. If you were interested in the value of `heads` at the halfway point of the program's execution, when 500 of 1,000 coin flips have been completed, you could instead just set a breakpoint on the line `print('Halfway done!')` ❷. To set a breakpoint, click the line number in the file editor. This should cause a red dot to appear, marking the breakpoint; see Figure 5-4.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart



```
print('Enter the second
second = input()
print('Enter the third n
third = input()
print('The sum is ' + fi
```

Figure 5-4: Setting a breakpoint causes a red dot (circled) to appear next to the line number.

Note that you wouldn't want to set a breakpoint on the `if` statement line, as the `if` statement executes on every single iteration through the loop. When you set the breakpoint on the code in the `if` statement, the debugger breaks only when the execution enters the `if` clause.

Now when you run the program under the debugger, it should start in a paused state at the first line, as usual, but if you click Continue, the program should run at full speed until it reaches the line with the breakpoint set on it. You can then click Continue, Step Over, Step In, or Step Out to continue as normal.

If you want to remove a breakpoint, click the line number again. The red dot will go away, and the debugger won't break on that line in the future.

Summary

Assertions, exceptions, logging, and the debugger are all valuable tools to find and prevent bugs in your program. Assertions with the Python `assert` statement are a good way to implement "sanity checks" that give you an early warning when a necessary condition doesn't hold true. Assertions are only for errors that the program shouldn't try to recover from, and they should fail fast. Otherwise, you should raise an exception.

An exception can be caught and handled by the `try` and `except` statements. The `logging` module is a good way to look into your code while it's running, and it is much more convenient to use than the `print()` function because of its different logging levels and its ability to log to a text file.

The debugger lets you step through your program one line at a time. Alternatively, you can run your program at normal speed and have the debugger pause execution whenever it reaches a line with a breakpoint set. Using the debugger, you can see the state of any variable's value at any point during the program's lifetime.

Accidentally introducing bugs into your code is a fact of life, no matter how many years of coding experience you have. These debugging tools and techniques will help you write programs that work.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Practice Questions

1. Write an `assert` statement that triggers an `AssertionError` if the variable `spam` is an integer less than `10`.
2. Write an `assert` statement that triggers an `AssertionError` if the variables `eggs` and `bacon` contain strings that are the same as each other, even if their cases are different. (That is, `'hello'` and `'hello'` are considered the same, as are `'goodbye'` and `'GOODbye'`.)
3. Write an `assert` statement that *always* triggers an `AssertionError`.
4. What two lines must your program have to be able to call `logging.debug()`?
5. What two lines must your program have to make `logging.debug()` send a logging message to a file named *programLog.txt*?
6. What are the five logging levels?
7. What line of code can you add to disable all logging messages in your program?
8. Why is using logging messages better than using `print()` to display the same message?
9. What are the differences between the Step Over, Step In, and Step Out buttons in the debugger?
10. After you click Continue, when will the debugger stop?
11. What is a breakpoint?
12. How do you set a breakpoint on a line of code in Mu?

Practice Program: Debugging Coin Toss

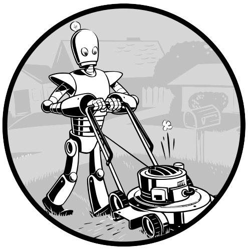
The following program is meant to be a simple coin toss guessing game. The player gets two guesses. (It's an easy game.) However, the program has multiple bugs in it. Run through the program a few times to find the bugs that keep the program from working correctly.

```
import random
guess = ''
while guess not in ('heads', 'tails'):
    print('Guess the coin toss! Enter heads or tails:')
    guess = input()
toss = random.randint(0, 1)  # 0 is tails, 1 is heads
if toss == guess:
    print('You got it!')
else:
    print('Nope! Guess again!')
    guess = input()
    if toss == guess:
        print('You got it!')
    else:
        print('Nope. You are really bad at this game.')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

6

LISTS



One more topic you'll need to understand before you can begin writing programs in earnest is the list data type and its cousin, the tuple. Lists and tuples can contain multiple values, which makes writing programs that handle large amounts of data easier. And since lists themselves can contain other lists, you can use them to arrange data into hierarchical structures.

In this chapter, I'll discuss the basics of lists. I'll also teach you about methods, which are functions that are tied to values of a certain data type. Then, I'll briefly cover the sequence data types (lists, tuples, and strings) and show their differences. In the next chapter, I'll introduce you to the dictionary data type.

The List Data Type

A *list* is a value that contains multiple values in an ordered sequence. The term *list value* refers to the list itself (which you can store in a variable or pass to a function, just like any other value), not the values inside the list value. A list value looks like this: `['cat', 'bat', 'rat', 'elephant']`. Just as string values use quotation marks to mark where the string begins and ends, a list begins with an opening square bracket and ends with a closing square bracket, `[]`.

We call values inside the list *items*. Items are separated with commas (that is, they are *comma-delimited*). For example, enter the following into the interactive shell:

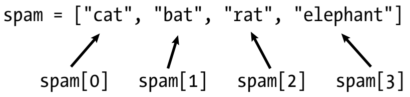
```
>>> [1, 2, 3]  # A list of three integers
[1, 2, 3]
>>> ['cat', 'bat', 'rat', 'elephant']  # A list of four strings
['cat', 'bat', 'rat', 'elephant']
>>> ['hello', 3.1415, True, None, 42]  # A list of several values
['hello', 3.1415, True, None, 42]
❶ >>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam
['cat', 'bat', 'rat', 'elephant']
```

The `spam` variable ❶ is assigned only one value: the list value. But the list value itself contains other values.

Note that the value `[]` is an empty list that contains no values, similar to `''`, the empty string.

Indexes

Say you have the list `['cat', 'bat', 'rat', 'elephant']` stored in a variable named `spam`. The Python code `spam[0]` would evaluate to `'cat'`, the code `spam[1]` would evaluate to `'bat'`, and so on. The integer inside the square brackets that follows the list is called an *index*. The first value in the list is at index `0`, the second value is at index `1`, the third value is at index `2`, and so on. Figure 6-1 shows a list value assigned to `spam`, along with the index expressions they'd evaluate to. Note that because the first index is `0`, the last index is the size of the list minus one. So, a list of four items has `3` as its last index.



```
spam = ["cat", "bat", "rat", "elephant"]
```

Figure 6-1: A list value stored in the variable spam, showing which value each index refers to

For an example of working with indexes, enter the following expressions into the interactive shell. We start by assigning a list to the variable `spam`:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0]
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
'cat'
>>> spam[1]
'bat'
>>> spam[2]
'rat'
>>> spam[3]
'elephant'
>>> ['cat', 'bat', 'rat', 'elephant'][3]
'elephant'
❶ >>> 'Hello, ' + spam[0]
❷ 'Hello, cat'
>>> 'The ' + spam[1] + ' ate the ' + spam[0] + '.'
'The bat ate the cat.'
```

Notice that the expression `'Hello, ' + spam[0]` ❶ evaluates to `'Hello, ' + 'cat'` because `spam[0]` evaluates to the string `'cat'`. This expression in turn evaluates to the string value `'Hello, cat'` ❷.

Python will give you an `IndexError` error message if you use an index that exceeds the number of values in your list value:

```
>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[10000]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    spam[10000]
IndexError: list index out of range
```

Lists can also contain other list values. You can access the values in these lists of lists using multiple indexes, like so:

```
>>> spam = [['cat', 'bat'], [10, 20, 30, 40, 50]]
>>> spam[0]
['cat', 'bat']
>>> spam[0][1]
'bat'
>>> spam[1][4]
50
```

The first index dictates which list value to use, and the second indicates the value within the list value. For example, `spam[0][1]` prints `'bat'`, the second value in the first list.

Negative Indexes

While indexes start at `0` and go up, you can also use negative integers for the index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[-1]  # Last index
'elephant'
```

```
>>> spam[-3]  # Third to last index
'bat'
>>> 'The ' + spam[-1] + ' is afraid of the ' + spam[-3] + '.'
'The elephant is afraid of the bat.'
```

The integer value `-1` refers to the last index in a list, the value `-2` refers to the second to last index in a list, and so on.

Slices

Just as an index can get a single value from a list, a *slice* can get several values from a list, in the form of a new list. We enter a slice between square brackets, like an index, but include two integers separated by a colon. Notice the difference between indexes and slices:

- `spam[2]` is a list with an index (one integer).
- `spam[1:4]` is a list with a slice (two integers).

In a slice, the first integer is the index where the slice starts. The second integer is the index where the slice ends. The list created from a slice will go up to, but will not include, the value at the second index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[0:4]
['cat', 'bat', 'rat', 'elephant']
>>> spam[1:3]
['bat', 'rat']
>>> spam[0:-1]
['cat', 'bat', 'rat']
```

As a shortcut, you can leave out one or both of the indexes on either side of the colon in the slice:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[:2]
['cat', 'bat']
>>> spam[1:]
['bat', 'rat', 'elephant']
>>> spam[:]
['cat', 'bat', 'rat', 'elephant']
```

Leaving out the first index is the same as using `0`, or the beginning of the list. Leaving out the second index is the same as using the length of the list, which will slice to the end of the list.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The len() Function

The len() function will return the number of values in a list value passed to it. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> len(spam)
3
```

This behavior is similar to how the function counts the number of characters in a string value.

Value Updates

Normally, a variable name goes on the left side of an assignment statement, as in spam = 42. However, you can also use an index of a list to change the value at that index:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam[1] = 'aardvark'
>>> spam
['cat', 'aardvark', 'rat', 'elephant']
>>> spam[2] = spam[1]
>>> spam
['cat', 'aardvark', 'aardvark', 'elephant']
>>> spam[-1] = 12345
>>> spam
['cat', 'aardvark', 'aardvark', 12345]
```

In this example, spam[1] = 'aardvark' means "Assign the value at index 1 in the list spam to the string 'aardvark'."

Concatenation and Replication

You can concatenate and replicate lists with the + and * operators, just like strings:

```
>>> [1, 2, 3] + ['A', 'B', 'C']
[1, 2, 3, 'A', 'B', 'C']
>>> ['X', 'Y', 'Z'] * 3
['X', 'Y', 'Z', 'X', 'Y', 'Z', 'X', 'Y', 'Z']
>>> spam = [1, 2, 3]
>>> spam = spam + ['A', 'B', 'C']
>>> spam
[1, 2, 3, 'A', 'B', 'C']
```

The + operator combines two lists to create a new list value, and the * operator combines a list and an integer value to replicate the list.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

del Statements

The del statement will delete values at an index in a list. All of the values in the list after the deleted value will be moved up one index. For example, enter the following into the interactive shell:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat', 'elephant']
>>> del spam[2]
>>> spam
['cat', 'bat']
```

The del statement can also operate on a simple variable to delete it, as if it were an "unassignment" statement. If you try to use the variable after deleting it, you'll get a NameError error because the variable no longer exists. In practice, you almost never need to delete simple variables, however, and the del statement is most useful for deleting values from lists.

Working with Lists

When you first begin writing programs, you may be tempted to create many individual variables to store a group of similar values. For example, if I wanted to store the names of my cats, I might think to write code like this:

```
cat_name_1 = 'Zophie'
cat_name_2 = 'Pooka'
cat_name_3 = 'Simon'
cat_name_4 = 'Lady Macbeth'
```

It turns out that this is a bad way to write code. For one thing, if the number of cats changes (and you can always have more cats), your program will never be able to store more cats than you have variables. These programs also contain a lot of duplicate or nearly identical code. To see this in practice, enter the following program into the file editor and save it as *allMyCats1.py*:

```
print('Enter the name of cat 1:')
cat_name_1 = input()
print('Enter the name of cat 2:')
cat_name_2 = input()
print('Enter the name of cat 3:')
cat_name_3 = input()
print('Enter the name of cat 4:')
cat_name_4 = input()
print('The cat names are:')
print(cat_name_1 + ' ' + cat_name_2 + ' ' + cat_name_3 + ' ' + cat_name_4)
```

Instead of using multiple, repetitive variables, you can use a single variable that contains a list value. For example, here's a new and improved version of the *allMyCats1.py* program. This new version uses a single list and can store any number of cats that the user enters. In a new file editor window, enter the following source code and save it as *allMyCats2.py*:

```
cat_names = []
while True:
    print('Enter the name of cat ' + str(len(cat_names) + 1) +
      ' (Or enter nothing to stop.):')
    name = input()
    if name == '':
        break
    cat_names = cat_names + [name]  # List concatenation
print('The cat names are:')
for name in cat_names:
    print('  ' + name)
```

When you run this program, the output will look something like this:

```
Enter the name of cat 1 (Or enter nothing to stop.):
Zophie
Enter the name of cat 2 (Or enter nothing to stop.):
Pooka
Enter the name of cat 3 (Or enter nothing to stop.):
Simon
Enter the name of cat 4 (Or enter nothing to stop.):
Lady Macbeth
Enter the name of cat 5 (Or enter nothing to stop.):

The cat names are:
  Zophie
  Pooka
  Simon
  Lady Macbeth
```

The benefit of using a list is that your data is now in a structure, so your program can process the data much more flexibly than it could with several repetitive variables.

for Loops and Lists

In Chapter 3, you learned about using `for` loops to execute a block of code a certain number of times. Technically, a `for` loop repeats the code block once for each item in a list value. For example, if you ran this code

```
for i in range(4):
    print(i)
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

the output of this program would be as follows:

```
0
1
2
3
```

This is because the return value from range(4) is a sequence value that Python considers to be similar to [0, 1, 2, 3]. The following program has the same output as the previous one:

```
for i in [0, 1, 2, 3]:
    print(i)
```

The previous for loop actually loops through its clause with the variable i set to a successive value in the [0, 1, 2, 3] list in each iteration.

A common Python technique is to use range(len(*some_list*)) with a for loop to iterate over the indexes of a list. For example, enter the following into the interactive shell:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for i in range(len(supplies)):
...     print('Index ' + str(i) + ' in supplies is: ' + supplies[i])
...
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

Using range(len(supplies)) in the previously shown for loop is handy because the code in the loop can access the index (as the variable i) and the value at that index (as supplies[i]). Best of all, range(len(supplies)) will iterate through all the indexes of supplies, no matter how many items the list contains.

The in and not in Operators

You can determine whether a value is or isn't in a list with the in and not in operators. Like other operators, in and not in occur in expressions and connect two values: a value to look for in a list and the list where it may be found. These expressions will evaluate to a Boolean value. To see how they work, enter the following into the interactive shell:

```
>>> 'howdy' in ['hello', 'hi', 'howdy', 'heyas']
True
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> 'cat' in spam
False
>>> 'howdy' not in spam
False
>>> 'cat' not in spam
True
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The following program lets the user enter a pet name and then checks whether the name is in a list of pets. Open a new file editor window, enter the following code, and save it as *myPets.py*:

```
my_pets = ['Zophie', 'Pooka', 'Fat-tail']
print('Enter a pet name:')
name = input()
if name not in my_pets:
    print('I do not have a pet named ' + name)
else:
    print(name + ' is my pet.')
```

The output may look something like this:

```
Enter a pet name:
Footfoot
I do not have a pet named Footfoot
```

Keep in mind that the `not in` operator is distinct from the Boolean `not` operator.

The Multiple Assignment Trick

The *multiple assignment trick* (technically called *tuple unpacking*) is a shortcut that lets you assign multiple variables with the values in a list in one line of code. So, instead of doing this

```
>>> cat = ['fat', 'gray', 'loud']
>>> size = cat[0]
>>> color = cat[1]
>>> disposition = cat[2]
```

you could enter this line of code:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition = cat
```

The number of variables and the length of the list must be exactly equal, or Python will give you a `ValueError`:

```
>>> cat = ['fat', 'gray', 'loud']
>>> size, color, disposition, name = cat
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    size, color, disposition, name = cat
ValueError: not enough values to unpack (expected 4, got 3)
```

This trick makes your code shorter and more readable than entering three separate lines of code.

List Item Enumeration

Instead of using the range(len(*some_list*)) technique with a for loop to obtain the integer index of the items in the list, you can call the enumerate() function. On each iteration of the loop, enumerate() will return two values: the index of the item in the list, and the item in the list itself. For example, this code is equivalent to the code in "Using for Loops with Lists" on page XX:

```
>>> supplies = ['pens', 'staplers', 'flamethrowers', 'binders']
>>> for index, item in enumerate(supplies):
...     print('Index ' + str(index) + ' in supplies is: ' + item)
...
Index 0 in supplies is: pens
Index 1 in supplies is: staplers
Index 2 in supplies is: flamethrowers
Index 3 in supplies is: binders
```

The enumerate() function is useful if you need both the item and the item's index in the loop's block.

Random Selection and Ordering

The random module has a couple of functions that accept lists for arguments. The random.choice() function will return a randomly selected item from the list. Enter the following into the interactive shell:

```
>>> import random
>>> pets = ['Dog', 'Cat', 'Moose']
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Cat'
>>> random.choice(pets)
'Dog'
```

You can consider random.choice(some_list) to be a shorter form of someList [random.randint(0, len(some_list) - 1].

The random.shuffle() function will reorder the items in a list in place. Enter the following into the interactive shell:

```
>>> import random
>>> people = ['Alice', 'Bob', 'Carol', 'David']
>>> random.shuffle(people)
>>> people
['Carol', 'David', 'Alice', 'Bob']
>>> random.shuffle(people)
>>> people
['Alice', 'David', 'Bob', 'Carol']
```

This function modifies the list in place, rather than returning a new list.

Augmented Assignment Operators

The + and * operators that work with strings also work with lists, so let's take a short detour to learn about augmented assignment operators. When assigning a value to a variable, you'll frequently use the variable itself. For example, after assigning `42` to the variable `spam`, you would increase the value in `spam` by `1` with the following code:

```
>>> spam = 42
>>> spam = spam + 1
>>> spam
43
```

As a shortcut, you can use the augmented assignment operator `+=` (which is the regular operator followed by one equal sign) to do the same thing:

```
>>> spam = 42
>>> spam += 1
>>> spam
43
```

There are augmented assignment operators for the `+`, `-`, `*`, `/`, and `%` operators, described in Table 6-1.

Table 6-1: The Augmented Assignment Operators

Augmented assignment statement	Equivalent assignment statement
`spam += 1`	`spam = spam + 1`
`spam -= 1`	`spam = spam - 1`
`spam *= 1`	`spam = spam * 1`
`spam /= 1`	`spam = spam / 1`
`spam %= 1`	`spam = spam % 1`

The `+=` operator can also do string and list concatenation, and the `*=` operator can do string and list replication. Enter the following into the interactive shell:

```
>>> spam = 'Hello,'
>>> spam += ' world!'  # Same as spam = spam + 'world!'
>>> spam
'Hello, world!'
>>> bacon = ['Zophie']
>>> bacon *= 3  # Same as bacon = bacon * 3
>>> bacon
['Zophie', 'Zophie', 'Zophie']
```

Like the multiple assignment trick, augmented assignment operators are a shortcut to make your code simpler and more readable.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Methods

A *method* is the same thing as a function, except it is *called on* a value. For example, if a list value were stored in `spam`, you would call the `index()` list method (which I'll explain shortly) on that list like so: `spam.index('hello')`. The method part comes after the value, separated by a period.

Each data type has its own set of methods. The list data type, for example, has several useful methods for finding, adding, removing, and otherwise manipulating values in a list. Think of a method as a function that is always associated with a value. In our `spam` list example, the function would hypothetically be `index(spam, 'hello')`. But since `index()` is a list method and not a function, we call `spam.index('hello')`. Calling `index()` on a list value is how Python knows `index()` is a list method. Let's learn about the list methods in Python.

Finding Values

List values have an `index()` method that can be passed a value. If that value exists in the list, the method will return the index of the value. If the value isn't in the list, then Python produces a `ValueError` error. Enter the following into the interactive shell:

```
>>> spam = ['hello', 'hi', 'howdy', 'heyas']
>>> spam.index('hello')
0
>>> spam.index('heyas')
3
>>> spam.index('howdy howdy howdy')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    spam.index('howdy howdy howdy')
ValueError: 'howdy howdy howdy' is not in list
```

When the list contains duplicates of the value, the method returns the index of its first appearance:

```
>>> spam = ['Zophie', 'Pooka', 'Fat-tail', 'Pooka']
>>> spam.index('Pooka')
1
```

Notice that `index()` returns `1`, not `3`.

Adding Values

To add new values to a list, use the `append()` and `insert()` methods. The `append()` method adds the argument to the end of the list:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.append('moose')
>>> spam
['cat', 'dog', 'bat', 'moose']
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The insert() method can insert a value at any index in the list. The first argument to insert() is the index of the new value, and the second argument is the new value to be inserted. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'bat']
>>> spam.insert(1, 'chicken')
>>> spam
['cat', 'chicken', 'dog', 'bat']
```

Notice that the code doesn't perform any assignment operation, such as spam = spam.append('moose') or spam = spam.insert(1, 'chicken'). The return value of append() and insert() is None, so you definitely wouldn't want to store it as the new variable value. Rather, these methods modify the list in place, a topic covered in more detail in "Mutable and Immutable Data Types" on page XX.

Methods belong to a single data type. The append() and insert() methods are list methods, and we can call them on list values only, not on values of other data types, such as strings or integers. To see what happens when we try to do so, enter the following into the interactive shell:

```
>>> eggs = 'hello'
>>> eggs.append('world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    eggs.append('world')
AttributeError: 'str' object has no attribute 'append'
>>> bacon = 42
>>> bacon.insert(1, 'world')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    bacon.insert(1, 'world')
AttributeError: 'int' object has no attribute 'insert'
```

Note the AttributeError error messages that show up.

Removing Values

The remove() method accepts a value to remove from the list on which it's called:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('bat')
>>> spam
['cat', 'rat', 'elephant']
```

Attempting to delete a value that doesn't exist in the list will result in a ValueError error. For example, enter the following into the interactive shell and notice the error it displays:

```
>>> spam = ['cat', 'bat', 'rat', 'elephant']
>>> spam.remove('chicken')
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    spam.remove('chicken')
ValueError: list.remove(x): x not in list
```

If the value appears multiple times in the list, the method will remove only the first instance of it:

```
>>> spam = ['cat', 'bat', 'rat', 'cat', 'hat', 'cat']
>>> spam.remove('cat')
>>> spam
['bat', 'rat', 'cat', 'hat', 'cat']
```

The del statement is useful when you know the index of the value you want to remove from the list, while the remove() method is useful when you know the value itself.

Sorting Values

You can sort lists of number values or lists of strings with the sort() method. For example, enter the following into the interactive shell:

```
>>> spam = [2, 5, 3.14, 1, -7]
>>> spam.sort()
>>> spam
[-7, 1, 2, 3.14, 5]
>>> spam = ['Ants', 'Cats', 'Dogs', 'Badgers', 'Elephants']
>>> spam.sort()
>>> spam
['Ants', 'Badgers', 'Cats', 'Dogs', 'Elephants']
```

The method returns the numbers in numerical order and the strings in alphabetical order. You can also pass True as the reverse keyword argument to sort the values in reverse order:

```
>>> spam.sort(reverse=True)
>>> spam
['Elephants', 'Dogs', 'Cats', 'Badgers', 'Ants']
```

Note three things about the sort() method. First, it sorts the list in place; don't try to capture the return value by writing code like spam = spam.sort().

Second, you can't sort lists that have both number values and string values in them, as Python doesn't know how to compare these values. Enter the following into the interactive shell and notice the TypeError error:

```
>>> spam = [1, 3, 2, 4, 'Alice', 'Bob']
>>> spam.sort()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    spam.sort()
TypeError: '<' not supported between instances of 'str' and 'int'
```

Third, `sort()` uses *ASCIIbetical order* rather than actual alphabetical order for sorting strings. This means uppercase letters come before lowercase letters, placing the lowercase *a* after the uppercase *Z*. For an example, enter the following into the interactive shell:

```
>>> spam = ['Alice', 'ants', 'Bob', 'badgers', 'Carol', 'cats']
>>> spam.sort()
>>> spam
['Alice', 'Bob', 'Carol', 'ants', 'badgers', 'cats']
```

If you need to sort the values in regular alphabetical order, pass `str.lower` for the `key` keyword argument in the `sort()` method call:

```
>>> spam = ['a', 'z', 'A', 'Z']
>>> spam.sort(key=str.lower)
>>> spam
['a', 'A', 'z', 'Z']
```

This argument causes the `sort()` function to treat all the items in the list as if they were lowercase without actually changing the values in the list.

Reversing Values

If you need to quickly reverse the order of the items in a list, you can call the `reverse()` list method. Enter the following into the interactive shell:

```
>>> spam = ['cat', 'dog', 'moose']
>>> spam.reverse()
>>> spam
['moose', 'dog', 'cat']
```

Like the `sort()` list method, `reverse()` doesn't return a list, which is why we write `spam.reverse()` instead of `spam = spam.reverse()`.

EXCEPTIONS TO INDENTATION RULES IN PYTHON

In most cases, the amount of indentation for a line of code tells Python what block it's in. There are some exceptions to this rule, however. For example, lists can actually span several lines in the source code file. The indentation of these lines doesn't matter; Python knows that the list isn't finished until it sees the ending square bracket. This means you can write code that looks like this:

```
spam = ['apples',
 'oranges',
                    'bananas',
'cats']
print(spam[0]) # Prints apples
```

(continued)

Of course, practically speaking, most people use Python's behavior to make their lists look pretty and readable, like the messages list in "A Short Program: Magic 8 Ball with a List" on page XX.

You can also split up a single instruction across multiple lines by ending each line with the *line continuation character* (\). Think of \ as saying, "This instruction continues on the next line." The indentation on the line after a \ line continuation isn't significant. For example, the following is valid Python code:

```
print('Four score and seven ' + \
      'years ago...')
```

These tricks are useful when you want to rearrange long lines of Python code to be a bit more readable.

Short-Circuiting Boolean Operators

Boolean operators have a subtle behavior that is easy to miss. Recall that if either of the values combined by an and operator is False, the entire expression is False, and if either value combined by an or operator is True, the entire expression is True. If I presented you with the expression False and spam, it doesn't matter whether the spam variable is True or False because the entire expression would be False either way. The same goes for True or spam; this evaluates to True no matter the value of spam.

Python (and many other programming languages) use this fact to optimize the code so that it runs a little faster by not examining the right-hand side of the Boolean operator at all. This shortcut is called *short-circuiting*. Most of the time, your program will behave the same way it would have if Python checked the entire expression (albeit a few microseconds faster). However, consider this short program, where we check whether the first item in a list is 'cat':

```
spam = ['cat', 'dog']
if spam[0] == 'cat':
    print('A cat is the first item.')
else:
    print('The first item is not a cat.')
```

As written, this program prints A cat is the first item. But if the list in spam is empty, the spam[0] code will cause an IndexError: list Index out of range error. To fix this, we'll adjust the if statement's condition to take advantage of short-circuiting:

```
spam = []
if len(spam) > 0 and spam[0] == 'cat':
    print('A cat is the first item.')
```

```
else:
    print('The first item is not a cat.')
```

This program never has an error, because if `len(spam) > 0` is `False` (that is, the list in `spam` is empty), then short-circuiting the `and` operator means that Python doesn't bother running the `spam[0] == 'cat'` code that would cause the `IndexError` error. Keep this short-circuiting behavior in mind when you write code that involves the `and` and `or` operators.

A Short Program: Magic 8 Ball with a List

Using lists, you can write a much more elegant version of Chapter 4's *magic8ball.py* program. Instead of several lines of nearly identical `elif` statements, you can create a single list that the code works with. Open a new file editor window and enter the following code. Save it as *magic8Ball2.py*:

```
import random

messages = ['It is certain',
    'It is decidedly so',
    'Yes definitely',
    'Reply hazy try again',
    'Ask again later',
    'Concentrate and ask again',
    'My reply is no',
    'Outlook not so good',
    'Very doubtful']

print('Ask a yes or no question:')
input('>')
print(messages[random.randint(0, len(messages) - 1)])
```

When you run it, you'll see that it works the same as the previous *magic8Ball.py* program.

The `random.randint(0, len(messages) - 1)` call produces a random number to use for the index, regardless of the size of `messages`. That is, you'll get a random number between `0` and the value of `len(messages) - 1`. The benefit of this approach is that you can easily add and remove strings to and from the `messages` list without changing other lines of code. If you later update your code, you'll have to change fewer lines, producing fewer chances for you to introduce bugs.

Selecting a random item from a list is common enough that Python has the `random.choice(messages)` function that does the same thing as `random.randint(0, len(messages) - 1)`.

Sequence Data Types

Lists aren't the only data types that represent ordered sequences of values. For example, strings and lists are actually similar if you consider a string to

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

be a "list" of single text characters. The Python sequence data types include lists, strings, range objects returned by `range()`, and tuples (explained in "The Tuple Data Type" on page XX). Many of the things you can do with lists can also be done with strings and other values of sequence types. To see this, enter the following into the interactive shell:

```
>>> name = 'Zophie'
>>> name[0]
'Z'
>>> name[-2]
'i'
>>> name[0:4]
'Zoph'
>>> 'Zo' in name
True
>>> 'z' in name
False
>>> 'p' not in name
False
>>> for i in name:
...     print('* * * ' + i + ' * * *')
...
* * * Z * * *
* * * o * * *
* * * p * * *
* * * h * * *
* * * i * * *
* * * e * * *
```

You can do all the same things with sequence values that you can do with lists: indexing, slicing, `for` loops, `len()`, and the `in` and `not in` operators.

Mutable and Immutable Data Types

But lists and strings differ in an important way. A list value is a *mutable* data type: you can add, remove, or change its values. However, a string is *immutable*: it cannot be changed. Trying to reassign a single character in a string results in a `TypeError` error, as you can see by entering the following into the interactive shell:

```
>>> name = 'Zophie a cat'
>>> name[7] = 'the'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    name[7] = 'the'
TypeError: 'str' object does not support item assignment
```

The proper way to "mutate" a string is to use slicing and concatenation to build a *new* string by copying from parts of the old string:

```
>>> name = 'Zophie a cat'
>>> new_name = name[0:7] + 'the' + name[8:12]
>>> name
```

```
'Zophie a cat'
>>> new_name
'Zophie the cat'
```

We used [0:7] and [8:12] to refer to the characters we don't wish to replace. Notice that the original 'Zophie a cat' string isn't modified, because strings are immutable.

Although a list value *is* mutable, the second line in the following code doesn't modify the list eggs:

```
>>> eggs = ['A', 'B', 'C']
>>> eggs = [ 'x', 'y', 'z']
>>> eggs
['x', 'y', 'z']
```

The list value in eggs isn't being changed here; rather, a new and entirely different list value (['x', 'y', 'z']) is replacing the old list value (['A', 'B', 'C']).

If you wanted to actually modify the original list in eggs to contain ['x', 'y', 'z'], you would have to use del statements and the append() method, like this:

```
>>> eggs = ['A', 'B', 'C']
>>> del eggs[2]
>>> del eggs[1]
>>> del eggs[0]
>>> eggs.append('x')
>>> eggs.append('y')
>>> eggs.append('z')
>>> eggs
['x', 'y', 'z']
```

In this example, the eggs variable ends up with the same list value it started with. It's just that this list has been changed (mutated) rather than overwritten. We call this *changing the list in place*.

Mutable versus immutable types may seem like a meaningless distinction, but "References" on page XX will explain the different behavior when calling functions with mutable arguments versus immutable arguments. First, however, let's find out about the tuple data type, which is an immutable form of the list data type.

The Tuple Data Type

There are only two differences between the *tuple* data type and the list data type. The first difference is that you write tuples using parentheses instead of square brackets. For example, enter the following into the interactive shell:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[0]
'hello'
>>> eggs[1:3]
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
(42, 0.5)
>>> len(eggs)
3
```

The second and primary way that tuples are different from lists is that tuples, like strings, are immutable: you can't modify, append, or remove their values. Enter the following into the interactive shell, and look at the resulting `TypeError` error message:

```
>>> eggs = ('hello', 42, 0.5)
>>> eggs[1] = 99
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    eggs[1] = 99
TypeError: 'tuple' object does not support item assignment
```

If you have only one value in your tuple, you can indicate this by placing a trailing comma after the value inside the parentheses. Otherwise, Python will think you've entered a value inside regular parentheses. (Unlike some other programming languages, it's fine to have a trailing comma after the last item in a list or tuple in Python.) Enter the following `type()` function calls into the interactive shell to see the distinction:

```
>>> type(('hello',))
<class 'tuple'>
>>> type(('hello'))
<class 'str'>
```

You can use tuples to convey to anyone reading your code that you don't intend for that sequence of values to change. If you need an ordered sequence of values that never changes, use a tuple. A second benefit of using tuples instead of lists is that, because they're immutable and their contents don't change, Python can implement optimizations that make code using tuples slightly faster than code using lists.

List and Tuple Type Conversion

Just as `str(42)` will return `'42'`, the string representation of the integer `42`, the functions `list()` and `tuple()` will return list and tuple versions of the values passed to them. Enter the following into the interactive shell, and notice that the return value is of a different data type than the value passed:

```
>>> tuple(['cat', 'dog', 5])
('cat', 'dog', 5)
>>> list(('cat', 'dog', 5))
['cat', 'dog', 5]
>>> list('hello')
['h', 'e', 'l', 'l', 'o']
```

Converting a tuple to a list is handy if you need a mutable version of a tuple value.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

References

A common metaphor is that variables are boxes that "store" values like strings and integers. However, this explanation is a simplification of what Python is actually doing. A better metaphor is that variables are paper name tags attached to values with string. Enter the following into the interactive shell:

```
❶ >>> spam = 42
❷ >>> eggs = spam
❸ >>> spam = 99
  >>> spam
  99
  >>> eggs
  42
```

When you assign 42 to the spam variable, you're actually creating the 42 value in the computer's memory and storing a *reference* to it in the spam variable. When you copy the value in spam and assign it to the variable eggs, you're copying the reference. Both the spam and eggs variables refer to the 42 value in the computer's memory. Using the name tag metaphor for variables, you've attached the spam name tag and the eggs name tag to the same 42 value. When you assign spam a new 99 value, you've changed what the spam name tag references. Figure 6-2 is a graphical depiction of the code.

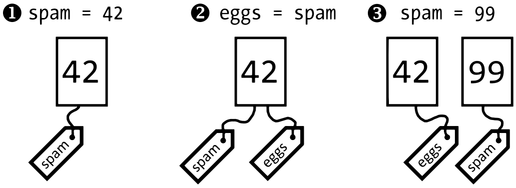


Figure 6-2: Variable assignment doesn't rewrite the value; it changes the reference.

The change doesn't affect eggs, which still refers to the 42 value.

But lists don't work this way, because list values can change; that is, lists are *mutable*. Here is code that will make this distinction easier to understand. Enter it into the interactive shell:

```
❶ >>> spam = [0, 1, 2, 3]
❷ >>> eggs = spam  # The reference, not the list, is being copied.
❸ >>> eggs[1] = 'Hello!'  # This changes the list value.
  >>> spam
  [0, 'Hello!', 2, 3]
  >>> eggs  # The cheese variable refers to the same list.
  [0, 'Hello!', 2, 3]
```

This code might look odd to you. It touched only the cheese list, but both the eggs and spam lists seem to have changed.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

When you create the list ❶, you assign a reference to it in the spam variable. But the next line copies only the list reference in spam to cheese, not the list value itself. There is still only one list, and spam and cheese now both refer to it. The reason there is only one underlying list is that the list itself was never actually copied. So, when you modify the first element of cheese ❷, you're modifying the same list that spam refers to. You can see this in Figure 6-3.

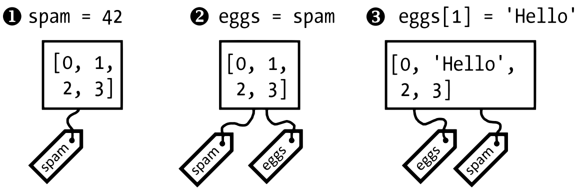


Figure 6-3: Because spam and eggs refer to the same list, changing one changes the other.

It becomes a bit more complicated, as lists also don't contain a sequence of values directly, but rather a sequence of references to values. I explain this further in "The copy() and deepcopy() Functions" on page XX.

Although Python variables technically contain references to values, people often casually say that the variable *contains* the value. But keep these two rules in mind:

- In Python, variables never contain values. They contain only references to values.
- In Python, the = assignment operator copies only references. It never copies values.

For the most part, you don't need to know these details, but at times, these simple rules have surprising effects, and you should understand exactly what Python is doing.

Arguments

References are particularly important for understanding how arguments get passed to functions. When a function is called, Python copies to the parameter variables the reference to the arguments. For mutable values like lists (and dictionaries, which I'll describe in Chapter 7), this means the code in the function modifies the original value in place. To see the consequences of this fact, open a new file editor window, enter the following code, and save it as *passingReference.py*:

```
def eggs(some_parameter):
    some_parameter.append('Hello')

spam = [1, 2, 3]
eggs(spam)
print(spam)  # Prints [1, 2, 3, 'Hello']
```

Notice that when you call eggs(), a return value doesn't assign a new value to spam. Instead, it directly modifies the list in place. When run, this program outputs [1, 2, 3, 'Hello'].

Even though spam and some_parameter contain separate references, they both refer to the same list. This is why the append('Hello') method call inside the function affects the list even after the function call has returned.

Keep this behavior in mind. Forgetting that Python handles list and dictionary variables in this way can lead to unexpected behavior and confusing bugs.

The copy() and deepcopy() Functions

Although passing around references is often the handiest way to deal with lists and dictionaries, if the function modifies the list or dictionary passed to it, you may not want these changes in the original list or dictionary value. To control this behavior, Python provides a module named copy that provides both the copy() and deepcopy() functions. The first of these, copy.copy(), can make a duplicate copy of a mutable value like a list or dictionary, not just a copy of a reference. Enter the following into the interactive shell:

```
>>> import copy
>>> spam = ['A', 'B', 'C']
>>> cheese = copy.copy(spam)  # Creates a duplicate copy of the list
>>> cheese[1] = 42  # Changes cheese
>>> spam  # The spam variable is unchanged.
['A', 'B', 'C']
>>> cheese  # The cheese variable is changed.
['A', 42, 'C']
```

Now the spam and cheese variables refer to separate lists, which is why only the list in cheese is modified when you assign 42 at index 1.

Just as variables *refer* to values rather than contain values, lists contain *references* to values rather than values themselves. You can see this in Figure 6-4.

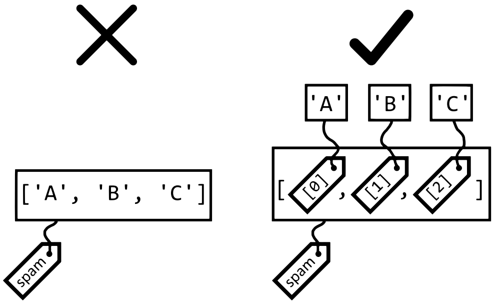


Figure 6-4: Lists don't contain values directly (left); they contain references to values (right).

If the list you need to copy contains lists, use the copy.deepcopy() function instead of copy.copy(). The copy.deepcopy() function will copy these inner lists as well.

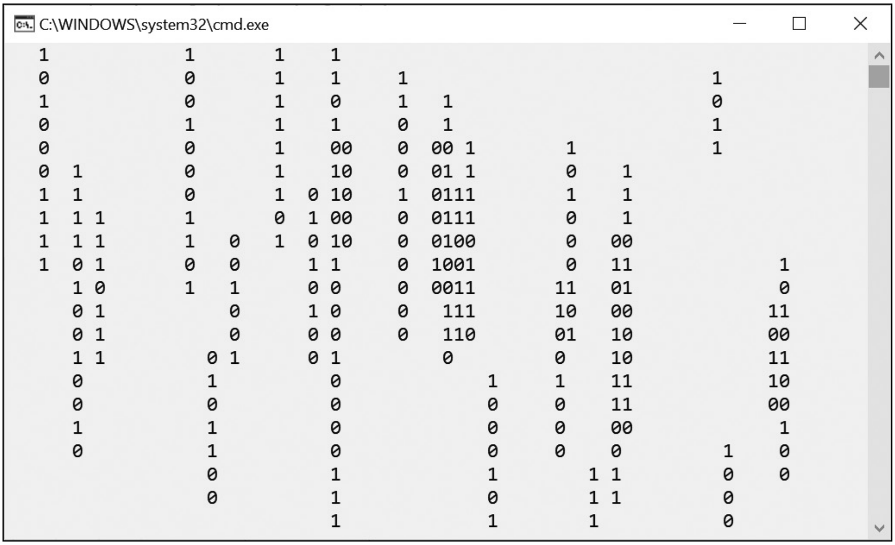
Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

A Short Program: The Matrix Screensaver

In the hacker science fiction film *The Matrix*, computer monitors display streams of glowing green numbers, like digital rain pouring down a glass window. The numbers may be meaningless, but they look cool. Just for fun, we can create our own Matrix screensaver in Python. Enter the following code into a new file and save it as *matrixscreensaver.py*:

```
import random, sys, time

WIDTH = 70  # The number of columns

try:
    # For each column, when the counter is 0, no stream is shown.
    # Otherwise, it acts as a counter for how many times a 1 or 0
    # should be displayed in that column.
    columns = [0] * WIDTH
    while True:
        # Loop over each column:
        for i in range(WIDTH):
            if random.random() < 0.02:
                # Restart a stream counter on this column.
                # The stream length is between 4 and 14 characters long.
                columns[i] = random.randint(4, 14)

            # Print a character in this column:
            if columns[i] == 0:
                # Change this ' '' to '.' to see the empty spaces:
                print(' ', end='')
            else:
                # Print a 0 or 1:
                print(random.choice([0, 1]), end='')
                columns[i] -= 1  # Decrement the counter for this column.
        print()  # Print a newline at the end of the row of columns.
        time.sleep(0.1)  # Each row pauses for one tenth of a second.
except KeyboardInterrupt:
    sys.exit()  # When Ctrl-C is pressed, end the program.
```

When you run this program, it produces streams of binary 1s and 0s, as in Figure 6-5.



```
1            1       1    1
0            0       1    1     1                           1
1            0       1    0     1   1                       0
0            1       1    1     0   1                       1
0            0       1    00    0  00 1        1            1
0  1         0       1    10    0  01 1        0    1
1  1         0       1  0 10    1  0111        1    1
1  1 1       1       0  1 00    0  0111        0    1
1  1 1       1   0   1  0 10    0  0100        0   00
1  0 1       0   0      1 1     0  1001        0   11             1
   1 0       1   1      0 0     0  0011       11   01             0
   0 1           0      1 0     0   111       10   00            11
   0 1           0      0 0     0   110       01   10            00
   1 1         0 1      0 1         0         0    10            11
   0           1          0             1     1    11            10
   0           0          0             0     0    11            00
   1           1          0             0     0    00             1
   0           1          0             0     0    0         1    0
               0          1             1        1 1         0    0
               0          1             0        1 1         0
                          1             1        1           0
```

Figure 6-5: The Matrix *screensaver program*

Like the previous spike and zigzag programs, this program creates a scrolling animation by printing rows of text inside an infinite loop that is stopped when the user presses CTRL-C. The main data structure in this program is the `columns` list, which holds 70 integers, one for each column of output. When an integer in `columns` is `0`, it prints an empty space for that column. When it's greater than `0`, it randomly prints a `0` or `1` and then decrements the integer. Once the integer is reduced to `0`, that column prints an empty space again. The program randomly sets the integers in `columns` to integers between `4` and `14` to produce streams of random binary 0s and 1s.

Let's take a look at each part of the program:

```
import random, sys, time

WIDTH = 70  # The number of columns
```

We import the `random` module for its `choice()` and `random()` functions, the `sys` module for its `exit()` function, and the `time` module for its `sleep()` function. We also set a variable named `WIDTH` to `70` so that the program produces output for 70 columns of characters. You're free to change this value to a larger or smaller integer based on the size of the window in which you run the program.

The `WIDTH` variable has an all-uppercase name because it's a constant variable. A *constant* is a variable that the code isn't supposed to change once set. Using constants allows you to write more readable code, such as `columns = [0] * WIDTH` instead of `columns = [0] * 70`, which may leave you wondering what the `70` is supposed to be when you reread the code later. In Python, nothing prevents you from changing a constant's value, but the uppercase name can remind the programmer not to do so.

The bulk of the program occurs inside a `try` block, which catches if the user presses CTRL-C to raise a `KeyboardInterrupt` exception:

```
try:
    # For each column, when the counter is 0, no stream is shown.
    # Otherwise, it acts as a counter for how many times a 1 or 0
    # should be displayed in that column.
    columns = [0] * WIDTH
```

The `columns` variable contains a list of `0` integers. The number of integers in this list is equal to the `WIDTH`. Each of these integers controls whether a column of the output window prints a stream of binary numbers or not:

```
    while True:
        # Loop over each column:
        for i in range(WIDTH):
            if random.random() < 0.02:
                # Restart a stream counter on this column.
                # The stream length is between 4 and 14 characters long.
                columns[i] = random.randint(4, 14)
```

We want this program to run forever, so we place it all inside an infinite `while True:` loop. Inside this loop is a `for` loop that iterates over each column of a single row. The loop variable `i` represents the indexes of columns; it begins at `0` and goes up to but does not include `WIDTH`. The value in `columns[0]` represents what should be printed in the leftmost column, `columns[1]` does so for the second column from the left, and so on.

For each column, there is a two percent chance that the integer at `columns[i]` is set to a number between `4` and `14`. We calculate this chance by comparing `random.random()` (a function that returns a random float between `0.0` and `1.0`) to `0.02`. If you want the streams to be denser or sparser, you can increase or decrease this number, respectively. We set the counter integers for each column to a random number between 4 and 14:

```
            # Print a character in this column:
            if columns[i] == 0:
                # Change this ' '' to '.' to see the empty spaces:
                print(' ', end='')
            else:
                # Print a 0 or 1:
                print(random.choice([0, 1]), end='')
                columns[i] -= 1  # Decrement the counter for this column.
```

Also inside the `for` loop, the program determines if it should print a random `0` or `1` binary number or an empty space. If `columns[i]` is `0`, it prints an empty space. Otherwise, it passes the list `[0, 1]` to the `random.choice()` function, which returns a random value from that list to print. The code also decrements the counter at `columns[i]` so that it gets closer to `0` and no longer prints binary numbers.

If you'd like to see the "empty" spaces the program prints, try changing the `' '` string to `'.'` and running the program again. The output should look like this:

```
............................1.........................................
................0...........1......................1..................
................1...........0................1.....0..................
............1...0...........0.....0..........1.....0..................
............1.1.1...........0.....0..........1.....1..1...............
............0.0.0...........0.....1.........00.....1..1...............
```

After the `else` block ends, the `for` loop block also ends:

```
        print()  # Print a newline at the end of the row of columns.
        time.sleep(0.1)  # Each row pauses for one tenth of a second.
except KeyboardInterrupt:
    sys.exit()  # When Ctrl-C is pressed, end the program.
```

The `print()` call after the `for` loop prints a newline, as the previous `print()` calls for each column pass the `end=''` keyword argument to prevent a newline from being printed after each column. For each row printed, the program introduces a tenth-of-a-second pause by calling `time.sleep(0.1)`.

The final part of the program is an `except` block that exits the program if the user pressed CTRL-C to raise a `KeyboardInterrupt` exception.

Summary

Lists are useful data types, as they allow you to write code that works on a modifiable number of values in a single variable. Later in this book, you'll see programs that use lists to do things that would otherwise be difficult or impossible.

A list is a sequence data type that is mutable, meaning that its contents can change. Tuples and strings, though also sequence data types, are immutable and cannot be changed. We can overwrite a variable that contains a tuple or string value with a new tuple or string value, which isn't the same thing as modifying the existing value in place—as, say, the `append()` or `remove()` method does on lists.

Variables don't store list values directly; they store references to lists. This is an important distinction when you're copying variables or passing lists as arguments in function calls. Because the value that is being copied is the list reference, be aware that any changes you make to the list might impact another variable in your program. You can use `copy()` or `deepcopy()` if you want to make changes to a list in one variable without modifying the original list.

Practice Questions

1. What is []?
2. How would you assign the value 'hello' as the third value in a list stored in a variable named spam? (Assume spam contains [2, 4, 6, 8, 10].)

   For the following three questions, assume spam contains the list ['a', 'b', 'c', 'd'].
3. What does spam[int(int('3' * 2) // 11)] evaluate to?
4. What does spam[-1] evaluate to?
5. What does spam[:2] evaluate to?

   For the following three questions, assume bacon contains the list [3.14, 'cat', 11, 'cat', True].
6. What does bacon.index('cat') evaluate to?
7. What does bacon.append(99) make the list value in bacon look like?
8. What does bacon.remove('cat') make the list value in bacon look like?
9. What are the operators for list concatenation and list replication?
10. What is the difference between the append() and insert() list methods?
11. What are two ways to remove values from a list?
12. Name a few ways that list values are similar to string values.
13. What is the difference between lists and tuples?
14. How do you write the tuple value that has just the integer value 42 in it?
15. How can you get the tuple form of a list value? How can you get the list form of a tuple value?
16. Variables that "contain" list values don't actually contain lists directly. What do they contain instead?
17. What is the difference between copy.copy() and copy.deepcopy()?

Practice Programs

For practice, write programs to do the following tasks.

Comma Code

Say you have a list value like this:

```
spam = ['apples', 'bananas', 'tofu', 'cats']
```

Write a function that takes a list value as an argument and returns a string with all the items separated by a comma and a space, with *and* inserted before the last item. For example, passing the previous spam list to the function would return 'apples, bananas, tofu, and cats'. But your function should be able to work with any list value passed to it. Be sure to test the case where an empty list [] is passed to your function.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Coin Flip Streaks

For this exercise, we'll try doing an experiment. If you flip a coin 100 times and write down an *H* for each heads and a *T* for each tails, you'll create a list that looks like *T T T T H H H H T T*. If you ask a human to make up 100 random coin flips, you'll probably end up with alternating heads-tails results like *H T H T H H T H T T*—which looks random (to humans), but isn't mathematically random. A human will almost never write down a streak of six heads or six tails in a row, even though it is highly likely to happen in truly random coin flips. Humans are predictably bad at being random.

Write a program to find out how often a streak of six heads or a streak of six tails comes up in a randomly generated list of 100 heads and tails. Your program should break up the experiment into two parts: the first part generates a list of 100 randomly selected `'H'` and `'T'` values, and the second part checks if there is a streak in it. Put all of this code in a loop that repeats the experiment 10,000 times so that you can find out what percentage of the coin flips contains a streak of six heads or six tails in a row. As a hint, the function call `random.randint(0, 1)` will return a `0` value 50 percent of the time and a `1` value the other 50 percent of the time.

You can start with the following template:

```
import random
number_of_streaks = 0
for experiment_number in range(10000):  # Run 100,000 experiments total.
    # Code that creates a list of 100 'heads' or 'tails' values

    # Code that checks if there is a streak of 6 heads or tails in a row

print('Chance of streak: %s%%' % (number_of_streaks / 100))
```

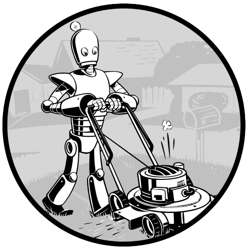
Of course, this is only an estimate, but 10,000 is a decent sample size. Some knowledge of mathematics could give you the exact answer and save you the trouble of writing a program, but programmers are notoriously bad at math.

To create a list, use a `for` loop that appends a randomly selected `'H'` or `'T'` to a list 100 times. To determine if there is a streak of six heads or six tails, create a slice like `some_list[i:i + 6]` (which contains the six items starting at index `i`) and then compare it to the list values `['H', 'H', 'H', 'H', 'H', 'H']` and `['T', 'T', 'T', 'T', 'T', 'T']`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

7

DICTIONARIES AND STRUCTURING DATA



This chapter covers the dictionary data type, which provides a flexible way to access and organize data. By combining dictionaries with your knowledge of lists from the previous chapter, you'll also learn how to create a data structure to model a chessboard.

The Dictionary Data Type

Like a list, a *dictionary* is a mutable collection of many values. But unlike indexes for lists, indexes for dictionaries can use many different data types, not just integers. These dictionary indexes are called *keys*, and a key with its associated value is called a *key-value pair*.

In code, a dictionary is entered between curly brackets ({}). Enter the following into the interactive shell:

```
>>> my_cat = {'size': 'fat', 'color': 'gray', 'age': 17}
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

This assigns a dictionary to the my_cat variable. This dictionary's keys are 'size', 'color', and 'age'. The values for these keys are 'fat', 'gray', and 17, respectively. You can access these values through their keys:

```
>>> my_cat['size']
'fat'
>>> 'My cat has ' + my_cat['color'] + ' fur.'
'My cat has gray fur.'
```

Using dictionaries, you can store multiple pieces of data about the same thing in a single variable. This my_cat variable contains three different strings describing my cat, and I can use it as an argument or return value in a function call, saving me from needing to create three separate variables.

Dictionaries can still use integer values as keys, just like lists use integers for indexes, but they don't have to start at 0 and can be any number:

```
>>> spam = {12345: 'Luggage Combination', 42: 'The Answer'}
>>> spam[12345]
'Luggage Combination'
>>> spam[42]
'The Answer'
>>> spam[0]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 0
```

Dictionaries have keys, not indexes. In this example, while the dictionary in spam has integer keys 12345 and 42, it doesn't have an index 0 through 41 like a list would.

Comparing Dictionaries and Lists

Unlike lists, items in dictionaries are unordered. The first item in a list named spam would be spam[0]. But there is no "first" item in a dictionary. While the order of items matters for determining whether two lists are the same, you can enter the key-value pairs of a dictionary in any order. Enter the following into the interactive shell:

```
>>> spam = ['cats', 'dogs', 'moose']
>>> bacon = ['dogs', 'moose', 'cats']
>>> spam == bacon  # The order of list items matters.
False
>>> eggs = {'name': 'Zophie', 'species': 'cat', 'age': '8'}
>>> ham = {'species': 'cat', 'age': '8', 'name': 'Zophie'}
>>> eggs == ham  # The order of dictionary key-value pairs doesn't matter.
True
```

The eggs and ham dictionaries are identical values even though we entered their key-value pairs in different orders. Because a dictionary isn't ordered, it isn't a sequence data type and can't be sliced like a list.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Trying to access a key that doesn't exist in a dictionary will result in a `KeyError` error message, much like a list's "out-of-range" `IndexError` error message. Enter the following into the interactive shell, and notice the error message that shows up because there is no `'color'` key:

```
>>> spam = {'name': 'Zophie', 'age': 7}
>>> spam['color']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    spam['color']
KeyError: 'color'
```

Though dictionaries aren't ordered, the fact that you can use arbitrary values as keys allows you to organize your data in powerful ways. Say you wanted your program to store data about your friends' birthdays. You can use a dictionary with the names as keys and the birthdays as values. Open a new file editor window and enter the following code, then save it as *birthdays.py*:

```
❶ birthdays = {'Alice': 'Apr 1', 'Bob': 'Dec 12', 'Carol': 'Mar 4'}

while True:
    print('Enter a name: (blank to quit)')
    name = input()
    if name == '':
        break

  ❷ if name in birthdays:
      ❸ print(birthdays[name] + ' is the birthday of ' + name)
    else:
        print('I do not have birthday information for ' + name)
        print('What is their birthday?')
        bday = input()
      ❹ birthdays[name] = bday
        print('Birthday database updated.')
```

The code creates an initial dictionary and stores it in `birthdays` ❶. You can see if the entered name exists as a key in the dictionary with the `in` keyword ❷, just as you did for lists. If the name is in the dictionary, you access the associated value using square brackets ❸; if not, you can add it using the same square bracket syntax combined with the assignment operator ❹.

When you run this program, it will look like this:

```
Enter a name: (blank to quit)
Alice
Apr 1 is the birthday of Alice
Enter a name: (blank to quit)
Eve
I do not have birthday information for Eve
What is their birthday?
Dec 5
Birthday database updated.
Enter a name: (blank to quit)
```

```
Eve
Dec 5 is the birthday of Eve
Enter a name: (blank to quit)
```

Of course, all the data you enter in this program is forgotten when the program terminates. You'll learn how to save data to files on the hard drive in Chapter 10.

Returning Keys and Values

Three dictionary methods will return list-like values of the dictionary's keys, values, or both keys and values: `keys()`, `values()`, and `items()`. The values returned by these methods aren't true lists: they can't be modified and don't have an `append()` method. But these data types (`dict_keys`, `dict_values`, and `dict_items`, respectively) *can* be used in `for` loops. To see how these methods work, enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for v in spam.values():
...     print(v)

red
42
```

Here, a `for` loop iterates over each of the values in the `spam` dictionary. A `for` loop can also iterate over the keys or both the keys and values:

```
>>> for k in spam.keys():
...     print(k)

color
age
>>> 'color' in spam.keys()
True
>>> 'age' not in spam.keys()
False
>>> 'red' in spam.values()
True
>>> for i in spam.items():
...     print(i)

('color', 'red')
('age', 42)
```

When you use the `keys()`, `values()`, and `items()` methods, a `for` loop can iterate over the keys, values, or key-value pairs in a dictionary, respectively, and you can use the `in` and `not in` operators to determine if a value exists as a key or value in the dictionary. Notice that the values in the `dict_items` value returned by the `items()` method are tuples of the key and value.

You can also use the in and not in operators with the dictionary value itself to check for the existence of a key. This is equivalent to using these operators with the keys() method:

```
>>> 'color' in spam
True
>>> 'color' in spam.keys()
True
```

If you want to get an actual list from one of these methods, pass its list-like return value to the list() function. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> spam.keys()  # Returns a list-like dict_keys value
dict_keys(['color', 'age'])
>>> list(spam.keys())  # Returns an actual list value
['color', 'age']
```

The list(spam.keys()) line takes the dict_keys value returned from keys() and passes it to list(), which then returns a list value of ['color', 'age'].

You can also use the multiple assignment trick in a for loop to assign the key and value to separate variables. Enter the following into the interactive shell:

```
>>> spam = {'color': 'red', 'age': 42}
>>> for k, v in spam.items():
...     print('Key: ' + str(k) + ' Value: ' + str(v))

Key: color Value: red
Key: age Value: 42
```

This code creates a dictionary with keys 'color' and 'age' whose values are 'red' and 42, respectively. The for loop iterates over the tuples returned by the items() method: ('color', 'red') and ('age', 42). The two variables, k and v, are assigned the first (the key) and second (the value) values from these tuples. The body of the loop prints out the k and v variables for each key-value pair.

While you can use many values for keys, you cannot use a list or dictionary as the key in a dictionary. These data types are *unhashable*, which is a concept beyond the scope of this book. If you need a list for a dictionary key, use a tuple instead.

Checking If a Key Exists

Checking whether a key exists in a dictionary before accessing that key's value can be tedious. Fortunately, dictionaries have a get() method that takes two arguments: the key of the value to retrieve and a fallback value to return if that key doesn't exist.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Enter the following into the interactive shell:

```
>>> picnic_items = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnic_items.get('cups', 0)) + ' cups.'
'I am bringing 2 cups.'
>>> 'I am bringing ' + str(picnic_items.get('eggs', 0)) + ' eggs.'
'I am bringing 0 eggs.'
```

Because there is no `'eggs'` key in the `picnic_items` dictionary, the `get()` method returns the default value `0`. Without using `get()`, the code would have caused an error message, such as in the following example:

```
>>> picnic_items = {'apples': 5, 'cups': 2}
>>> 'I am bringing ' + str(picnic_items['eggs']) + ' eggs.'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
    'I am bringing ' + str(picnic_items['eggs']) + ' eggs.'
KeyError: 'eggs'
```

Checking for the existence of a key before accessing the value for that key can prevent your programs from crashing with an error message.

Setting Default Values

You'll often have to set a value in a dictionary for a certain key only if that key doesn't already have a value. Your code might look something like this:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> if 'color' not in spam:
...     spam['color'] = 'black'
...
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

The `setdefault()` method offers a way to do this in one line of code. The first argument passed to the method is the key to check for, and the second argument is the value to set at that key if the key doesn't exist. If the key does exist, the `setdefault()` method returns the key's value. Enter the following into the interactive shell:

```
>>> spam = {'name': 'Pooka', 'age': 5}
>>> spam.setdefault('color', 'black')  # Sets 'color' key to 'black'
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
>>> spam.setdefault('color', 'white')  # Does nothing
'black'
>>> spam
{'name': 'Pooka', 'age': 5, 'color': 'black'}
```

The first time we call `setdefault()`, the dictionary in `spam` changes to `{'color': 'black', 'age': 5, 'name': 'Pooka'}`. The method returns the value `'black'` because this is now the value set for the key `'color'`. When `spam.setdefault('color', 'white')` is called next, the value for that key is *not* changed to `'white'`, because `spam` already has a key named `'color'`.

The `setdefault()` method is a nice shortcut to ensure that a key exists. Here is a short program that counts the number of occurrences of each letter in a string. Open the file editor window and enter the following code, saving it as *characterCount.py*:

```
message = 'It was a bright cold day in April, and the clocks were striking thirteen.'
count = {}

for character in message:
    count.setdefault(character, 0) ❶
    count[character] = count[character] + 1 ❷

print(count)
```

The program loops over each character in the `message` variable's string, counting how often each character appears. The `setdefault()` method call ❶ ensures that the key is in the `count` dictionary (with a default value of `0`) so that the program doesn't throw a `KeyError` error when `count[character] = count[character] + 1` is executed ❷. When you run this program, the output will look like this:

```
{'I': 1, 't': 6, ' ': 13, 'w': 2, 'a': 4, 's': 3, 'b': 1, 'r': 5, 'i': 6,
'g': 2, 'h': 3, 'c': 3, 'o': 2, 'l': 3, 'd': 3, 'y': 1, 'n': 4, 'A': 1,
'p': 1, ',': 1, 'e': 5, 'k': 2, '.': 1}
```

From the output, you can see that the lowercase letter *c* appears three times, the space character appears 13 times, and the uppercase letter *A* appears one time. This program will work no matter what string is inside the `message` variable, even if the string is millions of characters long!

Using Data Structures to Model Real-World Things

Even before the internet, it was possible to play a game of chess with someone on the other side of the world. Each player would set up a chessboard at their home, and then they would take turns mailing a postcard to each other describing their move. To do this, the players needed a way to unambiguously describe the state of the board and their moves.

In *algebraic chess notation*, the squares on the chessboard are identified by a number and letter coordinate, as in Figure 7-1.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

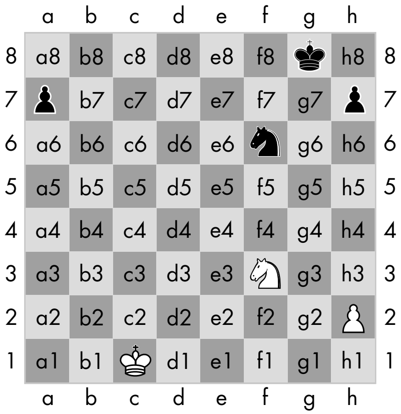


Figure 7-1: The coordinates of a chessboard in algebraic chess notation

The chess pieces use letters: *K* for king, *Q* for queen, *R* for rook, *B* for bishop, and *N* for knight. Describing a move requires specifying the letter of the piece and the coordinates of its destination. A pair of these moves describes what happens in a single turn (with white going first); for instance, the notation "*2. Nf3 Nc6*" indicates that white moved a knight to f3 and black moved a knight to c6 on the second turn of the game.

There's a bit more to algebraic notation than this, but the point is that you can unambiguously describe a game of chess without needing to be in front of a chessboard. Your opponent can even be on the other side of the world! In fact, you don't even need a physical chess set if you have a good memory: you can just read the mailed chess moves and update boards you have in your imagination.

Computers have good memories. A program on a modern computer can easily store billions of strings, such as `'2. Nf3 Nc6'`. This is how computers can play chess without having a physical chessboard. They model data to represent a chessboard, and you can write code to work with this model to simulate a chess game.

This is where lists and dictionaries can come in handy. For example, we could come up with our own notation so that the Python dictionary `{'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'}` could represent the chessboard in Figure 7-2.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

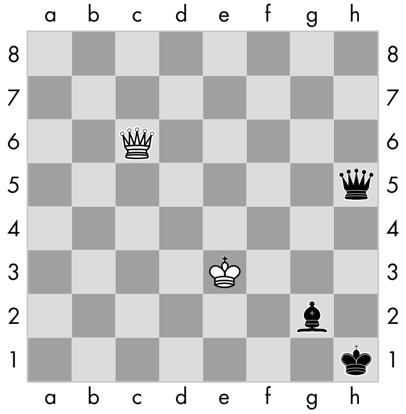


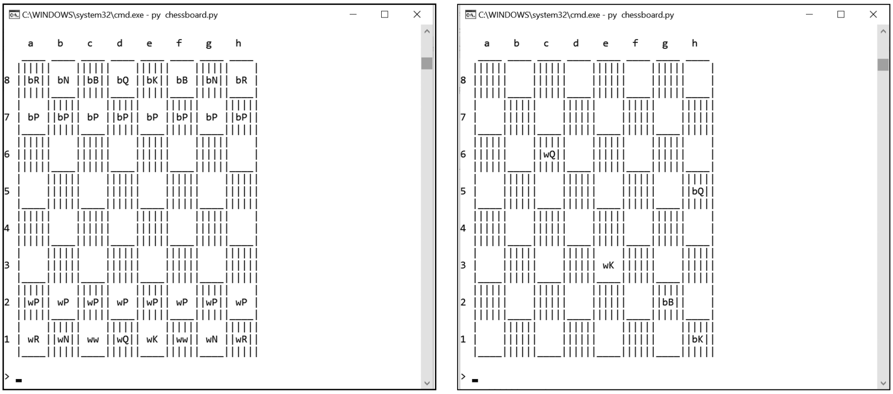
Figure 7-2: A chessboard modeled by the dictionary `{'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'}`

Let's use this data structure scheme to create our own interactive chessboard program.

Project 1: Interactive Chessboard Simulator

Even the earliest computers performed calculations far faster than any human, but back then, people considered chess a true demonstration of computational intelligence. We won't create our own chess-playing program here. (That would require its own book!) But we can create an interactive chessboard program with what we've discussed so far.

You don't need to know the rules of chess for this program. Just know that chess is played on an 8 × 8 board with white and black pieces called pawns, knights, bishops, rooks, queens, and kings. The upper-left and lower-right squares of the board should be white, and our program assumes the background of the output window is black (unlike the white background of a paper book). Our chessboard program is just a board with pieces on it; it doesn't even enforce the rules for how pieces move. We'll use text characters to "draw" a chessboard, such as the one shown in Figure 7-3.



C:\WINDOWS\system32\cmd.exe - py chessboard.py

```
    a    b    c    d    e    f    g    h
   ____ ____ ____ ____ ____ ____ ____ ____
  ||||||    ||||||    ||||||    ||||||    |
8 ||bR|| bN ||bB|| bQ ||bK|| bB ||bN|| bR |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
7 | bP ||bP|| bP ||bP|| bP ||bP|| bP ||bP||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
6 ||||||    ||||||    ||||||    ||||||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
5 |    ||||||    ||||||    ||||||    ||||||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
4 ||||||    ||||||    ||||||    ||||||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
3 |    ||||||    ||||||    ||||||    ||||||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
2 ||wP|| wP ||wP|| wP ||wP|| wP ||wP|| wP |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
1 | wR ||wN|| ww ||wQ|| wK ||ww|| wN ||wR||
  |____||||||____||||||____||||||____||||||

>
```

C:\WINDOWS\system32\cmd.exe - py chessboard.py

```
    a    b    c    d    e    f    g    h
   ____ ____ ____ ____ ____ ____ ____ ____
  ||||||    ||||||    ||||||    ||||||    |
8 ||||||    ||||||    ||||||    ||||||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
7 |    ||||||    ||||||    ||||||    ||||||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
6 ||||||    ||wQ||    ||||||    ||||||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
5 |    ||||||    ||||||    ||||||    ||bQ||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
4 ||||||    ||||||    ||||||    ||||||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
3 |    ||||||    |||||| wK ||||||    ||||||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
2 ||||||    ||||||    ||||||    ||bB||    |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
1 |    ||||||    ||||||    ||||||    ||bK||
  |____||||||____||||||____||||||____||||||

>
```

Figure 7-3: The non-graphical, text-based output of the chessboard program

Graphics would be nice and would make the pieces more readily identifiable, but we've captured all the information about the pieces without them. This text-based approach allows us to write the program with just the `print()` function and doesn't require us to install any sort of graphics library like Pygame (discussed in my book *Invent Your Own Computer Games with Python* [No Starch Press, 2016]) for our program.

First, we need to design a data structure that can represent a chessboard and any possible configuration of pieces on it. The example from the previous section works: the board is a Python dictionary with string keys `'a1'` to `'h8'` to represent squares on the board. Note that these strings are always two characters long. Also, the letter is always lowercase and comes before the number. This specificity is important; we'll use these details in the code.

To represent the pieces, we'll also use two-character strings, where the first letter is a lowercase `'w'` or `'b'` to indicate the white or black color, and the second letter is an uppercase `'P'`, `'N'`, `'B'`, `'R'`, `'Q'`, or `'K'` to represent the kind of piece. Figure 7-4 shows each piece, along with its string representation.

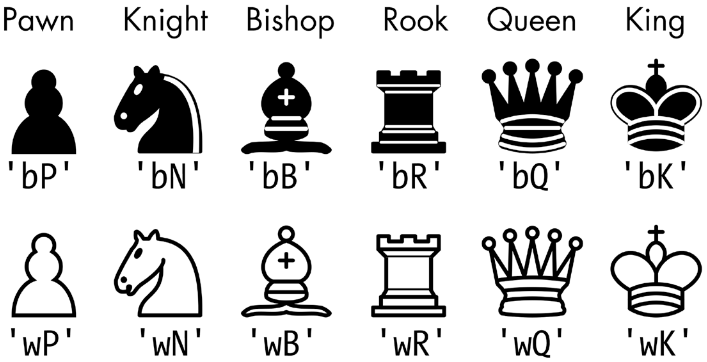


Figure 7-4: The two-character string representations for each chess piece

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The keys of the Python dictionary identify the squares of the board and the values identify the piece on that square. The absence of a key in the dictionary represents an empty square. A dictionary works well for storing this information: keys in a dictionary can be used only once, and squares on a chessboard can have only one piece on them at a time.

Step 1: Set Up the Program

The first part of the program imports the sys module for its `exit()` function and the `copy` module for its `copy()` function. At the start of the game, the white and black players have 16 pieces each. The `STARTING_PIECES` constant will hold a chessboard dictionary with all the proper pieces in their correct starting positions:

```
import sys, copy

STARTING_PIECES = {'a8': 'bR', 'b8': 'bN', 'c8': 'bB', 'd8': 'bQ',
'e8': 'bK', 'f8': 'bB', 'g8': 'bN', 'h8': 'bR', 'a7': 'bP', 'b7': 'bP',
'c7': 'bP', 'd7': 'bP', 'e7': 'bP', 'f7': 'bP', 'g7': 'bP', 'h7': 'bP',
'a1': 'wR', 'b1': 'wN', 'c1': 'ww', 'd1': 'wQ', 'e1': 'wK', 'f1': 'ww',
'g1': 'wN', 'h1': 'wR', 'a2': 'wP', 'b2': 'wP', 'c2': 'wP', 'd2': 'wP',
'e2': 'wP', 'f2': 'wP', 'g2': 'wP', 'h2': 'wP'}
```

(This code is a bit hard to type. You can copy and paste it from *https://autbor.com/3/chessboard.py*.) Whenever the program needs to reset the chessboard to the starting setup, it can copy `STARTING_PIECES` with the `copy.copy()` function.

Step 2: Create a Chessboard Template

The `BOARD_TEMPLATE` variable will contain a string that acts as a template for a chessboard. The program can insert the strings of individual pieces into it before printing. By using three double-quote characters in a row, we can create a *multiline string* that spans several lines of code. The multiline string ends with another three double-quote characters. This Python syntax is easier than trying to fit everything on a single line with `\n` escape characters. You'll learn more about multiline strings in the next chapter:

```
BOARD_TEMPLATE = """
    a    b    c    d    e    f    g    h

   ____ ____ ____ ____ ____ ____ ____ ____
  ||||||    ||||||    ||||||    ||||||    |
8 ||{}|| {} ||{}|| {} ||{}|| {} ||{}|| {} |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
7 | {} ||{}|| {} ||{}|| {} ||{}|| {} ||{}||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
6 ||{}|| {} ||{}|| {} ||{}|| {} ||{}|| {} |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
5 | {} ||{}|| {} ||{}|| {} ||{}|| {} ||{}||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
4 ||{}|| {} ||{}|| {} ||{}|| {} ||{}|| {} |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
3 | {} ||{}|| {} ||{}|| {} ||{}|| {} ||{}||
  |____||||||____||||||____||||||____||||||
  ||||||    ||||||    ||||||    ||||||    |
2 ||{}|| {} ||{}|| {} ||{}|| {} ||{}|| {} |
  ||||||____||||||____||||||____||||||____|
  |    ||||||    ||||||    ||||||    ||||||
1 | {} ||{}|| {} ||{}|| {} ||{}|| {} ||{}||
  |____||||||____||||||____||||||____||||||
"""
WHITE_SQUARE = '||'
BLACK_SQUARE = '  '
```

The pairs of curly brackets represent places in the string where we'll insert chess piece strings such as `'wR'` or `'bQ'`. If the square is empty, the program will insert the `WHITE_SQUARE` or `BLACK_SQUARE` string instead, which I'll explain in more detail when we discuss the `print_chessboard()` function.

Step 3: Print the Current Chessboard

We'll define a `print_chessboard()` function that accepts the chessboard dictionary, then prints a chessboard on the screen that reflects the pieces on this board. We'll call the `format()` string method on the `BOARD_TEMPLATE` string, passing the method a list of strings. The `format()` method returns a new string with the {} pairs in `BOARD_TEMPLATE` replaced by the strings in the passed-in list. You'll learn more about `format()` in the next chapter.

Let's take a look at the code in `print_chessboard()`:

```
def print_chessboard(board):
    squares = []
    is_white_square = True
    for y in '87654321':
        for x in 'abcdefgh':
            #print(x, y, is_white_square)  # DEBUG: Show coordinates
```

There are 64 squares on a chessboard and 64 {} pairs in the `BOARD_TEMPLATE` string. We must build up a list of 64 strings to replace these {} pairs. We store this list in the `squares` variable. The strings in this list represent either chess pieces, like `'wB'` and `'bQ'`, or empty squares. Depending on whether the empty square is white or black, we must use the `WHITE_SQUARE` string (`'||'`) or the `BLACK_SQUARE` string (`'  '`). We'll use a Boolean value in the `is_white_square` variable to keep track of which squares are white and which are black.

Two nested `for` loops will loop over all 64 squares on the board, starting with the upper-left square and going right to left, then top to bottom. The square in the upper left is a white square, so we'll start `is_white_square` as True. Remember that for loops can loop over the integers given by range(), the value in a list, or the individual characters in a string. In these two for loops, the y and x variables take on the characters in the strings '87654321' and 'abcdefgh', respectively. To see the order in which the code loops over the squares (along with the color of each square), uncomment the print(x, y), isWhiteSpace line of code before running the program.

The code inside the for loops builds up the squares list with the appropriate strings:

```
            if x + y in board.keys():
                squares.append(board[x + y])
            else:
                if is_white_square:
                    squares.append(WHITE_SQUARE)
                else:
                    squares.append(BLACK_SQUARE)
            is_white_square = not is_white_square
        is_white_square = not is_white_square

    print(BOARD_TEMPLATE.format(*squares))
```

The strings in the x and y loop variables concatenate together to form a two-character square string. For example, if x is 'a' and y is '8', then x + y evaluates to 'a8', and x + y in board.keys() checks if this string exists as a key in the chessboard dictionary. If it does, the code appends the chess piece string for that square to the end of the squares list.

If it does not, the code must append the blank square string in WHITE_SQUARE or BLACK_SQUARE, depending on the value in is_white_square. Once the code is done processing this chessboard square, it toggles the Boolean value in is_white_square to its opposite value (because the next square over will be the opposite color). The variable needs to be toggled again after finishing a row at the end of the outermost for loop.

After the loops have finished, the squares list contains 64 strings. However, the format() string method doesn't take a single list argument, but rather one string argument per {} pair to replace. The asterisk * next to the squares tells Python to pass the values in this list as individual arguments. This is a bit subtle, but imagine that you have a list spam = ['cat', 'dog', 'rat']. If you call print(spam), Python will print the list value, along with its square brackets, quotes, and commas. However, calling print(*spam) is equivalent to calling print('cat', 'dog', 'rat'), which simply prints cat dog rat. I call this *star syntax*.

The print_chessboard() function is written to work with the specific data structure we use to represent chessboards: a Python dictionary with keys of square strings, like 'a8', and values of pieces of strings, like 'bQ'. If we had designed our data structure differently, we'd have to write our function differently too. The print_chessboard() prints out a text-based representation of the board, but if we were using a graphics library like Pygame to render the chessboard, we could still use this Python dictionary to represent the chessboard configuration.

Step 4: Manipulate the Chessboard

Now that we have a way to represent a chessboard in a Python dictionary and a function to display a chessboard based on that dictionary, let's write code that moves pieces on the board by manipulating the keys and values of the dictionary. After the `print_chessboard()` function's `def` block, the main part of the program displays text explaining how to use the interactive chessboard program:

```
print('Interactive Chessboard')
print('by Al Sweigart al@inventwithpython.com')
print()
print('Pieces:')
print('  w - White, b - Black')
print('  P - Pawn, N - Knight, B - Bishop, R - Rook, Q - Queen, K - King')
print('Commands:')
print('  move e2 e4 - Moves the piece at e2 to e4')
print('  remove e2 - Removes the piece at e2')
print('  set e2 wP - Sets square e2 to a white pawn')
print('  reset - Resets pieces back to their starting squares')
print('  clear - Clears the entire board')
print('  fill wP - Fills entire board with white pawns.')
print('  quit - Quits the program')
```

The program can move pieces, remove pieces, set squares with a piece, reset the board, and clear the board by changing the chessboard dictionary:

```
main_board = copy.copy(STARTING_PIECES)
while True:
    print_chessboard(main_board)
    response = input('> ').split()
```

To begin, the `main_board` variable receives a copy of the `STARTING_PIECES` dictionary, which is a dictionary of all chess pieces in their standard starting positions. The execution enters an infinite loop that allows the user to enter commands. For example, if the user enters *move e2 e4* after `input()` is called, the `split()` method returns the list `['move', 'e2', 'e4']`, which the program then stores in the `response` variable. The first item in the `response` list, `response[0]`, will be the command the user wants to carry out:

```
    if response[0] == 'move':
        main_board[response[2]] = main_board[response[1]]
        del main_board[response[1]]
```

If the user enters something like *move e2 e4*, then `response[0]` is `'move'`. We can "move" a piece from one square to another by first copying the piece in `main_board` from the old square (in `response[1]`) to the new square (in `response[2]`). Then, we can delete the key-value pair for the old square in `main_board`. This has the effect of making it seem like the piece has moved (though we won't see this change until we call `print_chessboard()` again).

Our interactive chessboard simulator doesn't check if this is a valid move to make. It just carries out the commands given by the user. If the user enters something like *remove e2*, the program will set `response` to `['remove', 'e2']`:

```
    elif response[0] == 'remove':
        del main_board[response[1]]
```

By deleting the key-value pair at key `response[1]` from `main_board`, we make the piece disappear from the board. If the user enters something like *set e2 wP* to add a white pawn to e2, the program will set `response` to `['set', 'e2', 'wP']`:

```
    elif response[0] == 'set':
        main_board[response[1]] = response[2]
```

We can create a new key-value pair with the key `response[1]` and value `response[2]` in `main_board` to add this piece to the board. If the user enters *reset*, `response` is simply `['reset']`, and we can set the board to its starting configuration by copying the `STARTING_PIECES` dictionary to `main_board`:

```
    elif response[0] == 'reset':
        main_board = copy.copy(STARTING_PIECES)
```

If the user enters *clear*, `response` is simply `['clear']`, and we can remove all pieces from the board by setting `main_board` to an empty dictionary:

```
    elif response[0] == 'clear':
        main_board = {}
```

If the user enters *fill wP*, `response` is `['fill', 'wP']`, and we change all 64 squares to the string `'wP'`:

```
    elif response[0] == 'fill':
        for y in '87654321':
            for x in 'abcdefgh':
                main_board[x + y] = response[1]
```

The nested `for` loops will loop over every square, setting the `x + y` key to `response[1]`. There's no real reason to put 64 white pawns on a chessboard, but this command demonstrates how easy it is to manipulate the chessboard data structure however we want. Finally, the user can quit the program by entering *quit*:

```
    elif response[0] == 'quit':
        sys.exit()
```

After carrying out the command and modifying `main_board`, the execution jumps back to the start of the `while` loop to display the changed board and accept a new command from the user.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

This interactive chessboard program doesn't restrict what pieces you can place or move. It simply uses a dictionary as a representation of pieces on a chessboard and has one function for displaying such dictionaries on the screen in a way that looks like a chessboard. We can model all real-world objects or processes by designing data structures and writing functions to work with those data structures. If you'd like to see another example of modeling a game board with data structures, my other book, *The Big Book of Small Python Projects* (No Starch Press, 2021), has a working tic-tac-toe program.

Nested Dictionaries and Lists

As you model more complicated things, you may find you need dictionaries and lists that contain other dictionaries and lists. Lists are useful for holding an ordered series of values, and dictionaries are useful for associating keys with values. For example, here's a program that uses a dictionary to contain dictionaries of items guests are bringing to a picnic. The `total_brought()` function can read this data structure and calculate the total number of each item type. Enter the following code in a new program saved as *guestpicnic.py*:

```
all_guests = {'Alice': {'apples': 5, 'pretzels': 12},
             'Bob': {'ham sandwiches': 3, 'apples': 2},
             'Carol': {'cups': 3, 'apple pies': 1}}

def total_brought(guests, item):
    num_brought = 0
  ❶ for k, v in guests.items():
      ❷ num_brought = num_brought + v.get(item, 0)
    return num_brought

print('Number of things being brought:')
print(' - Apples         ' + str(total_brought(all_guests, 'apples')))
print(' - Cups           ' + str(total_brought(all_guests, 'cups')))
print(' - Cakes          ' + str(total_brought(all_guests, 'cakes')))
print(' - Ham Sandwiches ' + str(total_brought(all_guests, 'ham sandwiches')))
print(' - Apple Pies     ' + str(total_brought(all_guests, 'apple pies')))
```

Inside the `total_brought()` function, the `for` loop iterates over the key-value pairs in `guests` ❶. Inside the loop, the string of the guest's name is assigned to `k`, and the dictionary of picnic items they're bringing is assigned to `v`. If the item parameter exists as a key in this dictionary, its value (the quantity) is added to `num_brought` ❷. If it doesn't exist as a key, the `get()` method returns `0` to be added to `num_brought`.

The output of this program looks like this:

```
Number of things being brought:
- Apples 7
- Cups 3
- Cakes 0
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
 - Ham Sandwiches 3
 - Apple Pies 1
```

The number of items brought to a picnic may seem like such a simple thing to model that you wouldn't need to bother with writing a program to do it. But realize that this same `total_brought()` function could easily handle a dictionary that contains thousands of guests, each bringing thousands of different picnic items. In that case, having this information in a data structure, along with the `total_brought()` function, would save you a lot of time!

You can model things with data structures in whatever way you like, as long as the rest of the code in your program can work with the data model correctly. When you first begin programming, don't worry so much about the "right" way to model data. As you gain more experience, you may come up with more efficient models; the important thing is that the data model works for your program's needs.

Summary

You learned all about dictionaries in this chapter. Lists and dictionaries are values that can contain multiple values, including other lists and dictionaries. Dictionaries are useful because you can map one item (the key) to another item (the value), whereas lists simply contain a series of values in order. Code can access values inside a dictionary using square brackets just as with lists. Instead of integer indexes, dictionaries can have keys of a variety of data types: integers, floats, strings, or tuples. By organizing a program's values into data structures, you can create representations of real-world objects, such as the chessboard modeled in this chapter.

Practice Questions

1. What does the code for an empty dictionary look like?
2. What does a dictionary value with a key `'foo'` and a value `42` look like?
3. What is the main difference between a dictionary and a list?
4. What happens if you try to access `spam['foo']` if `spam` is `{'bar': 100}`?
5. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.keys()`?
6. If a dictionary is stored in `spam`, what is the difference between the expressions `'cat' in spam` and `'cat' in spam.values()`?
7. What is a shortcut for the following code?

```
if 'color' not in spam:
    spam['color'] = 'black'
```

8. What module and function can be used to "pretty-print" dictionary values?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Practice Programs

For practice, write programs to do the following tasks.

Chess Dictionary Validator

In this chapter, we used the dictionary value {'h1': 'bK', 'c6': 'wQ', 'g2': 'bB', 'h5': 'bQ', 'e3': 'wK'} to represent a chessboard. Write a function named isValidChessBoard() that takes a dictionary argument and returns True or False depending on whether the board is valid.

A valid board will have exactly one black king and exactly one white king. Each player can have at most 16 pieces, of which only eight can be pawns, and all pieces must be on a valid square from '1a' to '8h'. That is, a piece can't be on square '9z'. The piece names should begin with either a 'w' or a 'b' to represent white or black, followed by 'pawn', 'knight', 'bishop', 'rook', 'queen', or 'king'. This function should detect when a bug has resulted in an improper chessboard. (This isn't an exhaustive list of requirements, but it is close enough for this exercise.)

Fantasy Game Inventory

Say you're creating a medieval fantasy video game. The data structure to model the player's inventory is a dictionary whose keys are strings describing the item in the inventory and whose values are integers detailing how many of that item the player has. For example, the dictionary value {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12} means the player has one rope, six torches, 42 gold coins, and so on.

Write a function named display_inventory() that would take any possible "inventory" and display it like the following:

```
Inventory:
12 arrow
42 gold coin
1 rope
6 torch
1 dagger
Total number of items: 62
```

Hint: You can use a for loop to loop through all keys in a dictionary.

```
stuff = {'rope': 1, 'torch': 6, 'gold coin': 42, 'dagger': 1, 'arrow': 12}

def display_inventory(inventory):
    print("Inventory:")
    item_total = 0
    for k, v in inventory.items():
        # FILL THIS PART IN
    print("Total number of items: " + str(item_total))

display_inventory(stuff)
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

List-to-Dictionary Loot Conversion

Imagine that the same fantasy video game represents a vanquished dragon's loot as a list of strings, like this:

```
dragon_loot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
```

Write a function named `add_to_inventory(inventory, added_items)`. The `inventory` parameter is a dictionary representing the player's inventory (as in the previous project) and the `added_items` parameter is a list, like `dragon_loot`. The `add_to_inventory()` function should return a dictionary that represents the player's updated inventory. Note that the `added_items` list can contain multiples of the same item. Your code could look something like this:

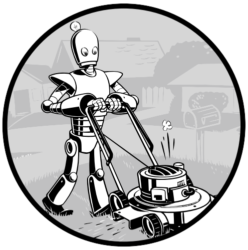
```
def add_to_inventory(inventory, added_items):
    # Your code goes here.

inv = {'gold coin': 42, 'rope': 1}
dragon_loot = ['gold coin', 'dagger', 'gold coin', 'gold coin', 'ruby']
inv = add_to_inventory(inv, dragon_loot)
display_inventory(inv)
```

The previous program (with your `display_inventory()` function from the previous project) would output the following:

```
Inventory:
45 gold coin
1 rope
1 ruby
1 dagger

Total number of items: 48
```

8

STRINGS AND TEXT EDITING



Text is one of the most common forms of data your programs will handle. You already know how to concatenate two string values with the + operator, but you can do much more than that, such as extract partial strings from string values, add or remove spacing, convert letters to lowercase or uppercase, and check that strings are formatted correctly. You can even write Python code to access the clipboard used for copying and pasting text.

In this chapter, you'll learn all of this and more. Then, you'll work through a programming project to automate the boring chore of adding bullet points to text.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Working with Strings

Let's look at some of the ways Python lets you write, print, and access strings in your code.

String Literals

While string *values* are stored in the program's memory, the string values that literally appear in our code are called *string literals*. Writing string literals in Python code seems straightforward: they begin and end with a single quotation mark, with the text of the string value in between. But how can you use quotes inside a string? Entering `'That is Alice's cat.'` won't work, because Python will think the string ends after `Alice` and will treat the rest (`s cat.'`) as invalid Python code. Fortunately, there are multiple ways to write string literals. The term *string* refers to a string value in the context of a running program and to a string literal when we're talking about entering Python source code.

Double Quotes

String literals can begin and end with double quotes as well as with single quotes. One benefit of using double quotes is that the string can have a single quote character in it. Enter the following into the interactive shell:

```
>>> spam = "That is Alice's cat."
```

Because the string begins with a double quote, Python knows that the single quote is part of the string and is not marking the end of the string. However, if you need to use both single quotes and double quotes in the string, you'll need to use escape characters.

Escape Characters

An *escape character* lets you use characters that are otherwise impossible to put into a string literal. An escape character consists of a backslash (\) followed by the character you want to add to the string. For example, `\'` is the escape character for a single quote and `\n` is the escape character for a newline character. (Despite consisting of two characters, it is commonly referred to as a singular escape character.) You can use this syntax inside a string that begins and ends with single quotes. To see how escape characters work, enter the following into the interactive shell:

```
>>> spam = 'Say hi to Bob\'s mother.'
```

Python knows that since the single quote in `Bob\'s` has a backslash, it is not a single quote meant to end the string value. The escape characters `\'` and `\"` let you put single quotes and double quotes inside your strings, respectively.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Table 8-1 lists the escape characters you can use.

Table 8-1: Escape Characters

Escape character	Prints as . . .
\'	Single quote
\"	Double quote
\t	Tab
\n	Newline (line break)
\\	Backslash

To practice using these, enter the following into the interactive shell:

```
>>> print("Hello there!\nHow are you?\nI\'m doing fine.")
Hello there!
How are you?
I'm doing fine.
```

Keep in mind that because the \ backslash begins an escape character, if you want an actual backslash in your string, you must use the \\ escape character.

Raw Strings

You can place an r before the beginning quotation mark of a string literal to make it a raw string literal. A *raw string* makes it easier to enter string values that have backslashes by ignoring all escape characters. For example, enter the following into the interactive shell:

```
>>> print(r'The file is in C:\Users\Alice\Desktop')
The file is in C:\Users\Alice\Desktop
```

Because this is a raw string, Python considers the backslash to be part of the string and not the start of an escape character:

```
>>> print('Hello...\n\n...world!')  # Without a raw string
Hello...

...world!
>>> print(r'Hello...\n\n...world!')  # With a raw string
Hello...\n\n...world!
```

Raw strings are helpful if your string values contain many backslashes, such as the strings used for Windows filepaths like r'C:\Users\Al\Desktop' or regular expression strings, which are described in the next chapter.

Multiline Strings

While you can use the \n escape character to insert a newline into a string, it's often easier to use multiline strings. A multiline string in Python begins and ends with either three single quotes or three double quotes. Any quotes, tabs, or newlines in between the "triple quotes" are considered part of the string. Python's indentation rules for blocks don't apply to lines inside a multiline string.

For example, open the file editor and enter the following:

```
print('''Dear Alice,

Can you feed Eve's cat this weekend?

Sincerely,
Bob''')
```

Save this program as *feedcat.py* and run it. The output will look like this:

```
Dear Alice,

Can you feed Eve's cat this weekend?

Sincerely,
Bob
```

Notice that the single quote character in `Eve's` doesn't need to be escaped. Escaping single and double quotes is optional in multiline strings:

```
print('Dear Alice,\n\nCan you feed Eve's cat this weekend?\n\nSincerely,\nBob')
```

This `print()` call prints identical text but doesn't use a multiline string.

Multiline Comments

While the hash character (`#`) marks the beginning of a comment for the rest of the line, a multiline string is often used for comments that span multiple lines:

```
"""This is a test Python program.
Written by Al Sweigart al@inventwithpython.com

This program was designed for Python 3, not Python 2.
"""

def say_hello():
    """This function prints hello.
    It does not return anything."""
    print('Hello!')
```

The multiline string in this example is perfectly valid Python code.

Indexes and Slices

Strings use indexes and slices the same way lists do. You can think of the string `'Hello, world!'` as a list and each character in the string as an item with a corresponding index and negative index:

'	H	e	l	l	o	,		w	o	r	l	d	!	'
	0	1	2	3	4	5	6	7	8	9	10	11	12	
	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1	

The space and exclamation mark are included in the character count, so `'Hello, world!'` is 13 characters long, from `H` at index 0 to `!` at index 12.

Enter the following into the interactive shell:

```
>>> greeting = 'Hello, world!'
>>> greeting[0]
'H'
>>> greeting [4]
'o'
>>> greeting[-1]
'!'
>>> greeting[0:5]
'Hello'
>>> greeting[:5]
'Hello'
>>> greeting[7:-1]
'world'
>>> greeting[7:]
'world!'
```

If you specify an index, you'll get the character at that position in the string. If you specify a range from one index to another, the starting index is included and the ending index is not. That's why, if `greeting` is `'Hello, world!'`, then `greeting[0:5]` evaluates to `'Hello'`. The substring you get from `greeting[0:5]` will include everything from `greeting[0]` to `greeting[4]`, leaving out the comma at index `5` and the space at index `6`. This is similar to how `range(5)` will cause a `for` loop to iterate up to, but not including, `5`.

Note that slicing a string doesn't modify the original string. You can capture a slice from one variable in a separate variable. Try entering the following into the interactive shell:

```
>>> greeting = 'Hello, world!'
>>> greeting_slice = greeting[0:5]
>>> greeting_slice
'Hello'
>>> greeting
'Hello, world!'
```

By slicing and storing the resulting substring in another variable, you can have both the whole string and the substring handy for quick, easy access.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The in and not in Operators

You can use the `in` and `not in` operators with strings just as you can with list values. An expression with two strings joined using `in` or `not in` will evaluate to a Boolean `True` or `False`. Enter the following into the interactive shell:

```
>>> 'Hello' in 'Hello, World'
True
>>> 'Hello' in 'Hello'
True
>>> 'HELLO' in 'Hello, World'
False
>>> '' in 'spam'
True
>>> 'cats' not in 'cats and dogs'
False
```

These expressions test whether the first string (including its capitalization) can be found within the second string.

F-Strings

Putting strings inside other strings is a common operation in programming. So far, we've been using the `+` operator and string concatenation to do this:

```
>>> name = 'Al'
>>> age = 4000
>>> 'Hello, my name is ' + name + '. I am ' + str(age) + ' years old.'
'Hello, my name is Al. I am 4000 years old.'
>>> In ten years I will be ' + str(age + 10)
'In ten years I will be 4010'
```

However, this requires a lot of tedious typing. A simpler approach is to use *f-strings*, which let you place variable names or entire expressions within a string. Like the `r` prefix in raw strings, f-strings have an `f` prefix before the starting quotation mark. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> f'My name is {name}. I am {age} years old.'
'My name is Al. I am 4000 years old.'
>>> f'In ten years I will be {age + 10}'
'In ten years I will be 4010'
```

Everything between the curly brackets (`{}`) is interpreted as if it were passed to `str()` and concatenated with the `+` operator in the middle of the string. If you need to use literal curly bracket characters in an f-string, use two curly brackets:

```
>>> name = 'Zophie'
>>> f'{name}'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
'Zophie'
>>> f'{{name}}'  # Use double curly brackets to include literal curly
brackets.
'{name}'
```

F-strings are a useful feature in Python, but the language only added them in version 3.6. In older Python code, you may run into alternative techniques.

F-String Alternatives: %s and format()

Versions of Python before 3.6 had other ways to put strings inside other strings. The first is *string interpolation*, in which strings included a %s format specifier that Python would replace with another string. For example, enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is %s. I am %s years old.' % (name, age)
'My name is Al. I am 4000 years old.'
>>> 'In ten years I will be %s' % (age + 10)
'In ten years I will be 4010'
```

Python will replace the first %s with the first value in the parentheses after the string, the second %s with the second string, and so on. This works just as well as f-strings if you need to insert only one or two strings, but f-strings tend to be more readable when you have several strings to insert.

The next way to put strings inside other strings is with the format() string method. You can use a pair of curly brackets to mark where to insert the strings, just like with string interpolation. Enter the following into the interactive shell:

```
>>> name = 'Al'
>>> age = 4000
>>> 'My name is {}. I am {} years old.'.format(name, age)
'My name is Al. I am 4000 years old.'
```

The format() method has a few more features than %s string interpolation. You can put the index integer (starting at 0) inside the curly brackets to note which of the arguments to format() should be inserted. This is helpful when inserting strings multiple times or out of order:

```
>>> name = 'Al'
>>> age = 4000
>>> '{1} years ago, {0} was born and named {0}.'.format(name, age)
'4000 years ago, Al was born and named Al.'
```

Most programmers prefer f-strings over these two alternatives, but you should learn them anyway, as you may run into them in existing code.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Useful String Methods

Several string methods analyze strings or create transformed string values. This section describes the methods you'll be using most often.

Changing the Case

The upper() and lower() string methods return a new string with all the letters in the original converted to uppercase or lowercase, respectively. Nonletter characters in the string remain unchanged. For example, enter the following into the interactive shell:

```
>>> spam = 'Hello, world!'
>>> spam = spam.upper()
>>> spam
'HELLO, WORLD!'
>>> spam = spam.lower()
>>> spam
'hello, world!'
```

Note that these methods don't change the string itself, but return new string values. If you want to change the original string, you have to call upper() or lower() on the string and then assign the new string to the variable that stored the original. This is why you must use spam = spam.upper() to change the string in spam instead of simply writing spam.upper(). (This is the same as if a variable eggs contains the value 10. Writing eggs + 3 doesn't change the value of eggs, but eggs = eggs + 3 does.)

The upper() and lower() methods are helpful if you need to make a case-insensitive comparison. For example, the strings 'great' and 'GREat' aren't equal to each other, but in the following small program, the user can enter Great, GREAT, or grEAT, because the code converts the string to lowercase:

```
print('How are you?')
feeling = input()
if feeling.lower() == 'great':
    print('I feel great too.')
else:
    print('I hope the rest of your day is good.')
```

When you run this program, it displays a question, and entering any variation on great, such as GREat, will give the output I feel great too. Adding code to your program to handle variations or mistakes in user input, such as inconsistent capitalization, will make your programs easier to use and less likely to fail:

```
How are you?
GREat
I feel great too.
```

The isupper() and islower() methods will return a Boolean True value if the string has at least one letter and all the letters are uppercase or lowercase, respectively. Otherwise, the method returns False. Enter the following into the interactive shell, and notice what each method call returns:

```
>>> spam = 'Hello, world!'
>>> spam.islower()
False
>>> spam.isupper()
False
>>> 'HELLO'.isupper()
True
>>> 'abc12345'.islower()
True
>>> '12345'.islower()
False
>>> '12345'.isupper()
False
```

Since the upper() and lower() string methods themselves return strings, you can call string methods on *those* returned string values as well:

```
>>> 'Hello'.upper()
'HELLO'
>>> 'Hello'.upper().lower()
'hello'
>>> 'Hello'.upper().lower().upper()
'HELLO'
>>> 'HELLO'.lower()
'hello'
>>> 'HELLO'.lower().islower()
True
```

Expressions that do this will look like a chain of method calls, as shown here.

Checking String Characteristics

Along with islower() and isupper(), several other string methods have names beginning with the word *is*. These methods return a Boolean value that describes the nature of the string. Here are some common is*X()* string methods:

isalpha() Returns True if the string consists only of letters and isn't blank

isalnum() Returns True if the string consists only of letters and numbers (alphanumerics) and isn't blank

isdecimal() Returns True if the string consists only of numeric characters and isn't blank

isspace() Returns True if the string consists only of spaces, tabs, and newlines and isn't blank

istitle() Returns True if the string consists only of words that begin with an uppercase letter followed by only lowercase letters

Enter the following into the interactive shell:

```
>>> 'hello'.isalpha()
True
>>> 'hello123'.isalpha()
False
>>> 'hello123'.isalnum()
True
>>> 'hello'.isalnum()
True
>>> '123'.isdecimal()
True
>>> '    '.isspace()
True
>>> 'This Is Title Case'.istitle()
True
```

The is*X()* string methods are helpful when you need to validate user input. For example, the following program repeatedly asks users for their age and a password until they provide valid input. Open a new file editor window and enter this program, saving it as *validateInput.py*:

```
while True:
    print('Enter your age:')
    age = input()
    if age.isdecimal():
        break
    print('Please enter a number for your age.')

while True:
    print('Select a new password (letters and numbers only):')
    password = input()
    if password.isalnum():
        break
    print('Passwords can only have letters and numbers.')
```

In the first while loop, we ask the user for their age and store their input in age. If age is a valid (decimal) value, we break out of this first while loop and move on to the second, which asks for a password. Otherwise, we inform the user that they need to enter a number and again ask them to enter their age. In the second while loop, we ask for a password, store the user's input in password, and break out of the loop if the input was alphanumeric. If it wasn't, we're not satisfied, so we tell the user the password needs to be alphanumeric and again ask them to enter a password.

When run, the program's output looks like this:

```
Enter your age:
forty two
Please enter a number for your age.
```

```
Enter your age:
42
Select a new password (letters and numbers only):
secr3t!
Passwords can only have letters and numbers.
Select a new password (letters and numbers only):
secr3t
```

Calling isdecimal() and isalnum() on variables, we're able to test whether the values stored in those variables are decimal or not and alphanumeric or not. Here, these tests help us reject the input forty two but accept 42, and reject secr3t! but accept secr3t.

Checking the Start or End of a String

The startswith() and endswith() methods return True if the string value on which they're called begins or ends (respectively) with the string passed to the method; otherwise, they return False. Enter the following into the interactive shell:

```
>>> 'Hello, world!'.startswith('Hello')
True
>>> 'Hello, world!'.endswith('world!')
True
>>> 'abc123'.startswith('abcdef')
False
>>> 'abc123'.endswith('12')
False
>>> 'Hello, world!'.startswith('Hello, world!')
True
>>> 'Hello, world!'.endswith('Hello, world!')
True
```

These methods are useful alternatives to the equals operator (==) if you need to check only whether the first or last part of the string, rather than the whole thing, is equal to another string.

Joining and Splitting Strings

The join() method is useful when you have a list of strings that need to be joined together into a single string value. We call the join() method on a string and pass it a list of strings, and it returns the concatenation of each string in the passed-in list. For example, enter the following into the interactive shell:

```
>>> ', '.join(['cats', 'rats', 'bats'])
'cats, rats, bats'
>>> ' '.join(['My', 'name', 'is', 'Simon'])
'My name is Simon'
>>> 'ABC'.join(['My', 'name', 'is', 'Simon'])
'MyABCnameABCisABCSimon'
```

Notice that the string on which `join()` is called is inserted between each string of the list argument. For example, when we call `join(['cats', 'rats', 'bats'])` on the `', '` string, it returns the string `'cats, rats, bats'`.

Remember that we call `join()` on a string value and pass it a list value. (It's easy to accidentally call it the other way around.) The `split()` method works the opposite way: we call it on a string value, and it returns a list of strings. Enter the following into the interactive shell:

```
>>> 'My name is Simon'.split()
['My', 'name', 'is', 'Simon']
```

By default, the method splits the string `'My name is Simon'` wherever it finds whitespace such as the space, tab, or newline characters. These whitespace characters aren't included in the strings in the returned list. You can pass a delimiter string to the `split()` method to specify a different string to split upon. For example, enter the following into the interactive shell:

```
>>> 'MyABCnameABCisABCSimon'.split('ABC')
['My', 'name', 'is', 'Simon']
>>> 'My name is Simon'.split('m')
['My na', 'e is Si', 'on']
```

A common use of `split()` is to split a multiline string along the newline characters. For example, enter the following into the interactive shell:

```
>>> spam = '''Dear Alice,
... There is a milk bottle in the fridge
... that is labeled "Milk Experiment."
...
... Please do not drink it.
... Sincerely,
... Bob'''
...
>>> spam.split('\n')
['Dear Alice,', 'There is a milk bottle in the
fridge', 'that is labeled "Milk Experiment."', '', 'Please do not drink it.',
'Sincerely,', 'Bob']
```

Passing `split()` the argument `'\n'` lets us split the multiline string stored in `spam` along the newlines and return a list in which each item corresponds to one line of the string.

Justifying and Centering Text

The `rjust()` and `ljust()` string methods return a padded version of the string on which they're called, with spaces inserted to justify the text. The first argument to both methods is an integer length for the justified string. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(10)
'     Hello'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> 'Hello'.rjust(20)
'               Hello'
>>> 'Hello, World'.rjust(20)
'        Hello, World'
>>> 'Hello'.ljust(10)
'Hello     '
```

The code `'Hello'.rjust(10)` says that we want to right-justify `'Hello'` in a string of total length `10`. `'Hello'` is five characters, so five spaces will be added to its left, giving us a string of 10 characters with `'Hello'` right-justified.

An optional second argument to `rjust()` and `ljust()` will specify a fill character other than a space character. Enter the following into the interactive shell:

```
>>> 'Hello'.rjust(20, '*')
'***************Hello'
>>> 'Hello'.ljust(20, '-')
'Hello---------------'
```

The `center()` string method works like `ljust()` and `rjust()` but centers the text, rather than justifying it to the left or right. Enter the following into the interactive shell:

```
>>> 'Hello'.center(20)
'       Hello        '
>>> 'Hello'.center(20, '=')
'=======Hello========'
```

Now the printed text is centered.

Removing Whitespace

Sometimes you may want to strip off whitespace characters (spaces, tabs, and newlines) from the left side, right side, or both sides of a string. The `strip()` string method will return a new string without any whitespace characters at the beginning or end, while the `lstrip()` and `rstrip()` methods will remove whitespace characters from the left and right ends, respectively. Enter the following into the interactive shell:

```
>>> spam = '    Hello, World    '
>>> spam.strip()
'Hello, World'
>>> spam.lstrip()
'Hello, World    '
>>> spam.rstrip()
'    Hello, World'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Optionally, a string argument will specify which characters on the ends to strip. Enter the following into the interactive shell:

```
>>> spam = 'SpamSpamBaconSpamEggsSpamSpam'
>>> spam.strip('ampS')
'BaconSpamEggs'
```

Passing `strip()` the argument `'ampS'` will tell it to strip occurrences of `a`, `m`, `p`, and `S` from the ends of the string stored in `spam`. The order of the characters in the string passed to `strip()` doesn't matter: `strip('ampS')` will do the same thing as `strip('mapS')` or `strip('Spam')`.

Numeric Code Points of Characters

Computers store information as *bytes* (strings of binary numbers), which means we need to be able to convert text to numbers. Because of this requirement, every text character has a corresponding numeric value called a *Unicode code point*. For example, the numeric code point is `65` for `'A'`, `52` for `'4'`, and `33` for `'!'`. You can use the `ord()` function to get the code point of a one-character string, and the `chr()` function to get the one-character string of an integer code point. Enter the following into the interactive shell:

```
>>> ord('A')
65
>>> ord('4')
52
>>> ord('!')
33
>>> chr(65)
'A'
```

These functions are useful when you need to order or perform a mathematical operation on characters:

```
>>> ord('B')
66
>>> ord('A') < ord('B')
True
>>> chr(ord('A'))
'A'
>>> chr(ord('A') + 1)
'B'
```

There is more to Unicode and code points than this, but those details are beyond the scope of this book. If you'd like to know more, I recommend watching or reading Ned Batchelder's 2012 PyCon talk, "Pragmatic Unicode, or How Do I Stop the Pain?" at *https://nedbatchelder.com/text/unipain.html.*

When strings are written to a file or sent over the internet, the conversion from text to bytes is called *encoding*. There are several Unicode encoding standards, but the most popular is UTF-8. If you ever need to choose a Unicode encoding, 'utf-8' is the correct answer 99 percent of the time. Tom Scott has a Computerphile video titled "Characters, Symbols and the Unicode Miracle" at *https://youtu.be/MijmeoH9LT4* that explains UTF-8 in particular.

Copying and Pasting Strings

The pyperclip module has copy() and paste() functions that can send text to and receive text from your computer's clipboard. Sending the output of your program to the clipboard will make it easy to paste it into an email, a word processor, or some other software.

The pyperclip module doesn't come with Python. To install it, follow the directions for installing third-party modules in Appendix A. After installing pyperclip, enter the following into the interactive shell:

```
>>> import pyperclip
>>> pyperclip.copy('Hello, world!')
>>> pyperclip.paste()
'Hello, world!'
```

Of course, if something outside your program changes the clipboard contents, the paste() function will return that other value. For example, if I copied this sentence to the clipboard and then called paste(), it would look like this:

```
>>> pyperclip.paste()
'For example, if I copied this sentence to the clipboard and then called
paste(), it would look like this:'
```

The clipboard is an excellent way to enter and receive large amounts of text without having to type it when prompted by an input() call. For example, say you want a program to turn text into aLtErNaTiNg uppercase and lowercase letters. You can copy the text you want to alternate to the clipboard, and then run this program, which takes this text and puts the alternating-case text on the clipboard. Enter the following code into a file named *alternatingText.py*:

```
import pyperclip

text = pyperclip.paste()  # Get the text off the clipboard.
alt_text = ''  # This string holds the alternating case.
make_uppercase = False
for character in text:
    # Go through each character and add it to alt_text:
    if make_uppercase:
        alt_text += character.upper()
    else:
        alt_text += character.lower()

    # Set make_uppercase to its opposite value:
    make_uppercase = not make_uppercase
```

```
pyperclip.copy(alt_text)  # Put the result on the clipboard.
print(alt_text)  # Print the result on the screen too.
```

If you copy some text to the clipboard (for instance, this sentence) and run this program, the output and clipboard contents become this:

```
iF YoU CoPy sOmE TeXt tO ThE ClIpBoArD (fOr iNsTaNcE, tHiS SeNtEnCe) AnD
 RuN ThIs pRoGrAm, ThE OuTpUt aNd cLiPbOaRd cOnTeNtS BeCoMe ThIs:
```

The pyperclip module's ability to interact with the clipboard gives you a straightforward way to input and output text to and from your programs.

Project 2: Add Bullets to Wiki Markup

When editing a Wikipedia article, you can create a bulleted list by putting each list item on its own line and placing a star in front of it. But say you have a really large list that you want to add bullet points to. You could type those stars at the beginning of each line, one by one. Or you could automate this task with a short Python script.

The *bulletPointAdder.py* script will get the text from the clipboard, add a star and space to the beginning of each line, and then paste this new text to the clipboard. For example, say I copied the following text (for the Wikipedia article "List of Lists of Lists") to the clipboard:

```
Lists of animals
Lists of aquarium life
Lists of biologists by author abbreviation
Lists of cultivars
```

Then, if I ran the *bulletPointAdder.py* program, the clipboard would contain the following:

```
* Lists of animals
* Lists of aquarium life
* Lists of biologists by author abbreviation
* Lists of cultivars
```

This star-prefixed text is ready to be pasted into a Wikipedia article as a bulleted list.

Step 1: Copy and Paste from the Clipboard

You want the *bulletPointAdder.py* program to do the following:

- Paste text from the clipboard.
- Do something to it.
- Copy the new text to the clipboard.

Manipulating the text is a little tricky, but copying and pasting are pretty straightforward: they just involve the pyperclip.copy() and pyperclip.paste()

functions. For now, let's write the part of the program that calls these functions. Enter the following, saving the program as *bulletPointAdder.py*:

```
import pyperclip
text = pyperclip.paste()

# TODO: Separate lines and add stars.

pyperclip.copy(text)
```

The TODO comment is a reminder that you should complete this part of the program eventually. The next step is to actually implement that piece of the program.

Step 2: Separate the Lines of Text

The call to pyperclip.paste() returns all the text on the clipboard as one big string. If we used the "List of Lists of Lists" example, the string stored in text would look like this:

```
'Lists of animals\nLists of aquarium life\nLists of biologists by author
abbreviation\nLists of cultivars'
```

The \n newline characters in this string cause it to be displayed with multiple lines when printed or pasted from the clipboard. There are many "lines" in this one string value. You want to add a star to the start of each of these lines.

You could write code that searches for each \n newline character in the string and then adds the star just after that. But it would be easier to use the split() method to return a list of strings, one for each line in the original string, and then add the star to the front of each string in the list.

Edit your program so that it looks like the following:

```
import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):  # Loop through all indexes in the "lines" list.
    lines[i] = '* ' + lines[i] # Add a star to each string in the "lines" list.

pyperclip.copy(text)
```

We split the text along its newlines to get a list in which each item is one line of the text. We store the list in lines and then loop through the items in lines. For each line, we add a star and a space to the start of the line. Now each string in lines begins with a star.

Step 3: Join the Modified Lines

The lines list now contains modified lines that start with stars. But pyperclip.copy() is expecting a single string value, not a list of string values. To make this single string value, pass lines into the join() method to get a single string joined from the list's strings:

```
import pyperclip
text = pyperclip.paste()

# Separate lines and add stars.
lines = text.split('\n')
for i in range(len(lines)):  # Loop through all indexes in the "lines" list.
    lines[i] = '* ' + lines[i]  # Add a star to each string in the "lines" list.
text = '\n'.join(lines)
pyperclip.copy(text)
```

When this program is run, it replaces the text on the clipboard with text that has stars at the start of each line. Now the program is complete, and you can try running it with text copied to the clipboard.

Even if you don't need to automate this specific task, you might want to automate some other kind of text manipulation, such as removing trailing spaces from the end of lines or converting text to uppercase or lowercase. Whatever your needs, you can use the clipboard for input and output.

A Short Program: Pig Latin

Pig latin is a silly made-up language that alters English words. If a word begins with a vowel, the word *yay* is added to the end of it. If a word begins with a consonant or consonant cluster (like *ch* or *gr*), that consonant or consonant cluster is moved to the end of the word and followed by *ay*.

Let's write a pig latin program that will output something like this:

```
Enter the English message to translate into pig latin:
My name is AL SWEIGART and I am 4,000 years old.
Ymay amenay isyay ALYAY EIGARTSWAY andyay Iyay amyay 4,000 yearsyay oldyay.
```

This program works by altering a string using the methods introduced in this chapter. Enter the following source code into the file editor, and save the file as *pigLat.py*:

```
# English to pig latin
print('Enter the English message to translate into pig latin:')
message = input()

VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')

pigLatin = [] # A list of the words in pig latin
for word in message.split():
    # Separate the non-letters at the start of this word:
```

```
    prefixNonLetters = ''
    while len(word) > 0 and not word[0].isalpha():
        prefixNonLetters += word[0]
        word = word[1:]
    if len(word) == 0:
        pigLatin.append(prefixNonLetters)
        continue

    # Separate the non-letters at the end of this word:
    suffixNonLetters = ''
    while not word[-1].isalpha():
        suffixNonLetters += word[-1] + suffixNonLetters
        word = word[:-1]

    # Remember if the word was in uppercase or title case:
    wasUpper = word.isupper()
    wasTitle = word.istitle()

    word = word.lower() # Make the word lowercase for translation.

    # Separate the consonants at the start of this word:
    prefixConsonants = ''
    while len(word) > 0 and not word[0] in VOWELS:
        prefixConsonants += word[0]
        word = word[1:]

    # Add the pig latin ending to the word:
    if prefixConsonants != '':
        word += prefixConsonants + 'ay'
    else:
        word += 'yay'

    # Set the word back to uppercase or title case:
    if wasUpper:
        word = word.upper()
    if wasTitle:
        word = word.title()

    # Add the non-letters back to the start or end of the word.
    pigLatin.append(prefixNonLetters + word + suffixNonLetters)

# Join all the words back together into a single string:
print(' '.join(pigLatin))
```

Let's look at this code line by line, starting at the top:

```
# English to pig latin
print('Enter the English message to translate into pig latin:')
message = input()

VOWELS = ('a', 'e', 'i', 'o', 'u', 'y')
```

First, we ask the user to enter the English text to translate into pig latin. Also, we create a constant that holds every lowercase vowel (and *y*) as a tuple of strings. We'll use this variable later.

Next, we create the pigLatin variable to store the words as we translate them into pig latin:

```
pigLatin = [] # A list of the words in pig latin
for word in message.split():
    # Separate the non-letters at the start of this word:
    prefixNonLetters = ''
    while len(word) > 0 and not word[0].isalpha():
        prefixNonLetters += word[0]
        word = word[1:]
    if len(word) == 0:
        pigLatin.append(prefixNonLetters)
        continue
```

We need each word to be its own string, so we call message.split() to get a list of the words as separate strings. The string 'My name is AL SWEIGART and I am 4,000 years old.' would cause split() to return ['My', 'name', 'is', 'AL', 'SWEIGART', 'and', 'I', 'am', '4,000', 'years', 'old.'].

We also need to remove any non-letters from the start and end of each word so that strings like 'old.' translate to 'oldyay.' instead of 'old.yay'. We save these non-letters to a variable named prefixNonLetters.

```
    # Separate the non-letters at the end of this word:
    suffixNonLetters = ''
    while not word[-1].isalpha():
        suffixNonLetters += word[-1] + suffixNonLetters
        word = word[:-1]
```

A loop that calls isalpha() on the first character in the word determines whether we should remove a character from a word and concatenate it to the end of prefixNonLetters. If the entire word is made of non-letter characters, like '4,000', we can simply append it to the pigLatin list and continue to the next word to translate. We also need to save the non-letters at the end of the word string. This code is similar to the previous loop.

Next, we make sure the program remembers if the word was in uppercase or title case so that we can restore it after translating the word to pig latin:

```
    # Remember if the word was in uppercase or title case:
    wasUpper = word.isupper()
    wasTitle = word.istitle()

    word = word.lower() # Make the word lowercase for translation.
```

For the rest of the code in the for loop, we'll work on a lowercase version of word.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

To convert a word like *sweigart* to *eigart-sway*, we need to remove all of the consonants from the beginning of word:

```
    # Separate the consonants at the start of this word:
    prefixConsonants = ''
    while len(word) > 0 and not word[0] in VOWELS:
        prefixConsonants += word[0]
        word = word[1:]
```

We use a loop similar to the loop that removed the non-letters from the start of word, except now we're pulling off consonants and storing them in a variable named prefixConsonants.

If there were any consonants at the start of the word, they're now in prefixConsonants, and we should concatenate that variable and the string 'ay' to the end of word. Otherwise, we can assume word begins with a vowel and we only need to concatenate 'yay':

```
    # Add the pig latin ending to the word:
    if prefixConsonants != '':
        word += prefixConsonants + 'ay'
    else:
        word += 'yay'
```

Recall that we set word to its lowercase version with word = word.lower(). If word was originally in uppercase or title case, this code will convert word back to its original case:

```
    # Set the word back to uppercase or title case:
    if wasUpper:
        word = word.upper()
    if wasTitle:
        word = word.title()
```

At the end of the for loop, we append the word, along with any non-letter prefix or suffix it originally had, to the pigLatin list:

```
    # Add the non-letters back to the start or end of the word.
    pigLatin.append(prefixNonLetters + word + suffixNonLetters)

# Join all the words back together into a single string:
print(' '.join(pigLatin))
```

After this loop finishes, we combine the list of strings into a single string by calling the join() method, then pass this single string to print() to display our pig latin on the screen.

Summary

Text is a common form of data, and Python comes with many helpful string methods to process the text stored in string values. You'll make use of indexing, slicing, and string methods in almost every Python program you write.

The programs you are writing now don't seem too sophisticated; they don't have graphical user interfaces (GUIs) with images and colorful text. So far, you're displaying text with `print()` and letting the user enter text with `input()`. However, the user can quickly enter large amounts of text through the clipboard. This ability provides a useful avenue for writing programs that manipulate massive amounts of text. These text-based programs might not have flashy windows or graphics, but they can get a lot of useful work done quickly.

Another way to manipulate large amounts of text is reading and writing files directly off the hard drive. You'll learn how to do this with Python in Chapter 10.

That just about covers all the basic concepts of Python programming! You'll continue to learn new concepts throughout the rest of this book, but you now know enough to start writing some useful programs that can automate tasks. If you'd like to see a collection of short, simple Python programs built from the basic concepts you've learned so far, you can read my other book, *The Big Book of Small Python Projects* (No Starch Press, 2021). Try copying the source code for each program by hand, and then make modifications to see how they affect the behavior of the program. Once you understand how the program works, try re-creating the program yourself from scratch. You don't need to re-create the source code exactly; just focus on what the program does rather than how it does it.

You might not think you have enough Python knowledge to do things such as download web pages, update spreadsheets, or send text messages, but that's where Python modules come in! These modules, written by other programmers, provide functions that make it easy for you to do all these things. In the next chapter, you'll learn how to write real programs to do useful automated tasks.

Practice Questions

1. What are escape characters?
2. What do the `\n` and `\t` escape characters represent?
3. How can you put a \ backslash character in a string?
4. The string value `"Howl's Moving Castle"` is a valid string. Why isn't it a problem that the single quote character in the word `Howl's` isn't escaped?
5. If you don't want to put `\n` in your string, how can you write a string with newlines in it?
6. What do the following expressions evaluate to?
    - `'Hello, world!'[1]`
    - `'Hello, world!'[0:5]`
    - `'Hello, world!'[:5]`
    - `'Hello, world!'[3:]`

7. What do the following expressions evaluate to?
   - `'Hello'.upper()`
   - `'Hello'.upper().isupper()`
   - `'Hello'.upper().lower()`
8. What do the following expressions evaluate to?
   - `'Remember, remember, the fifth of November.'.split()`
   - `'-'.join('There can be only one.'.split())`
9. What string methods can you use to right-justify, left-justify, and center a string?
10. How can you trim whitespace characters from the beginning or end of a string?

Practice Program: Table Printer

For practice, write a function named `printTable()` that takes a list of lists of strings and displays it in a well-organized table with each column right-justified. Assume that all the inner lists will contain the same number of strings. For example, the value could look like this:

```
tableData = [['apples', 'oranges', 'cherries', 'banana'],
             ['Alice', 'Bob', 'Carol', 'David'],
             ['dogs', 'cats', 'moose', 'goose']]
```

Your `printTable()` function would print the following:

```
  apples Alice  dogs
 oranges   Bob  cats
cherries Carol moose
  banana David goose
```

Hint: Your code will first have to find the longest string in each of the inner lists so that the whole column can be wide enough to fit all the strings. You can store the maximum width of each column as a list of integers. The `printTable()` function can begin with `colWidths = [0] * len(tableData)`, which will create a list containing the same number of `0` values as the number of inner lists in `tableData`. That way, `colWidths[0]` can store the width of the longest string in `tableData[0]`, `colWidths[1]` can store the width of the longest string in `tableData[1]`, and so on. You can then find the largest value in the `colWidths` list to find out what integer width to pass to the `rjust()` string method.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

PART II

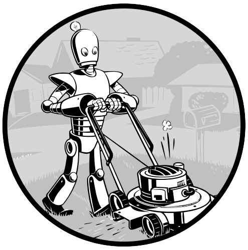
AUTOMATING TASKS

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

9

TEXT PATTERN MATCHING WITH REGULAR EXPRESSIONS



You may be familiar with the process of searching for text by pressing CTRL-F and entering the words you're looking for. *Regular expressions* go one step further: they allow you to specify a pattern of text to search for. You may not know a business's exact phone number, but if you live in the United States or Canada, you know it will consist of a three-digit area code, followed by a hyphen, then three more digits, another hyphen, and four more digits. This is how you, as a human, know a phone number when you see it: 415-555-1234 is a phone number, but $4,155,551,234 is not.

We recognize all sorts of other text patterns every day: email addresses have @ symbols in the middle, US Social Security numbers have nine digits and two hyphens, website URLs often have periods and forward slashes,

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

news headlines use title case, and social media hashtags begin with # and contain no spaces, to give some examples.

Regular expressions are helpful, but few nonprogrammers know about them, even though most modern text editors and word processors have find-and-replace features that can search based on regular expressions. Regular expressions are huge time-savers, not just for software users but also for programmers. In fact, in the *Guardian* article "Here's What ICT Should Really Teach Kids: How to Do Regular Expressions," tech writer Cory Doctorow argues that we should be teaching regular expressions before we teach programming:

> Knowing [regular expressions] can mean the difference between solving a problem in 3 steps and solving it in 3,000 steps. When you're a nerd, you forget that the problems you solve with a couple keystrokes can take other people days of tedious, error-prone work to slog through.

In this chapter, you'll start by writing a program to find text patterns *without* using regular expressions and then learn how to use regular expressions to make the code simpler. I'll show you basic matching with regular expressions, then move on to some more powerful features, such as string substitution and creating your own character classes. You'll also learn how to use the Humre module, which offers plain-English substitutes for regular expressions' cryptic symbol-based syntax.

Finding Text Patterns Without Regular Expressions

Say you want to find a US phone number in a string; you're looking for three numbers, a hyphen, three numbers, a hyphen, and four numbers. Here's an example: 415-555-4242.

Let's write a function named `is_phone_number()` to check whether a string matches this pattern and return either `True` or `False`. Open a new file editor tab and enter the following code, then save the file as *isPhoneNumber.py*:

```
def is_phone_number(text):
  ❶ if len(text) != 12:  # Phone numbers have exactly 12 characters.
        return False
    for i in range(0, 3):  # The first three characters must be numbers.
      ❷ if not text[i].isdecimal():
            return False
  ❸ if text[3] != '-':  # The fourth character must be a dash.
        return False
    for i in range(4, 7): # The next three characters must be numbers.
      ❹ if not text[i].isdecimal():
            return False
  ❺ if text[7] != '-':  # The eighth character must be a dash.
        return False
    for i in range(8, 12):  # The next four characters must be numbers.
      ❻ if not text[i].isdecimal():
            return False
  ❼ return True
```

```
print('Is 415-555-4242 a phone number?', is_phone_number('415-555-4242'))
print(is_phone_number('415-555-4242'))
print('Is Moshi moshi a phone number?', is_phone_number('Moshi moshi'))
print(is_phone_number('Moshi moshi'))
```

When this program is run, the output looks like this:

```
Is 415-555-4242 a phone number?
True
Is Moshi moshi a phone number?
False
```

The `is_phone_number()` function has code that does several checks to determine whether the string in `text` is a valid phone number. If any of these checks fail, the function returns `False`. First, the code checks that the string is exactly 12 characters long ❶. Then, it checks that the area code (that is, the first three characters in `text`) consists of only numeric characters ❷ by calling the `isdecimal()` string method. The rest of the function checks that the string follows the pattern of a phone number: the number must have the first hyphen after the area code ❸, three more numeric characters ❹, another hyphen ❺, and finally, four more numeric characters ❻. If the program execution manages to get past all the checks, it returns `True` ❼.

Calling `is_phone_number()` with the argument `'415-555-4242'` will return `True`. Calling `is_phone_number()` with `'Moshi moshi'` will return `False`; the first test fails because `'Moshi moshi'` is not 12 characters long.

If you wanted to find a phone number within a larger string, you would have to add even more code to locate the pattern. Replace the last four `print()` function calls in *isPhoneNumber.py* with the following:

```
message = 'Call me at 415-555-1011 tomorrow. 415-555-9999 is my office.'
for i in range(len(message)):
  ❶ segment = message[i:i+12]
  ❷ if is_phone_number(segment):
         print('Phone number found: ' + segment)
print('Done')
```

When this program is run, the output will look like this:

```
Phone number found: 415-555-1011
Phone number found: 415-555-9999
Done
```

On each iteration of the `for` loop, a new segment of 12 characters from `message` is assigned to the variable `segment` ❶. For example, on the first iteration, `i` is `0`, and `segment` is assigned `message[0:12]` (that is, the string `'Call me at 4'`). On the next iteration, `i` is `1`, and `segment` is assigned `message[1:13]` (the string `'all me at 41'`). In other words, on each iteration of the `for` loop, `segment` takes on the following values

```
'Call me at 4'
'all me at 41'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
'll me at 415'
'l me at 415-'
```

and so on, until its last value is `'s my office.'`

The loop's code passes `segment` to `is_phone_number()` to check whether it matches the phone number pattern ❷, and if so, it prints the segment. Once it has finished going through `message`, we print `Done`.

While the string in `message` is short in this example, the program would run in less than a second even if it were millions of characters long. A similar program that finds phone numbers using regular expressions would also run in less than a second; however, regular expressions make writing these programs much quicker.

Finding Text Patterns with Regular Expressions

The previous phone number–finding program works, but it uses a lot of code to do something limited. The `is_phone_number()` function is 17 lines but can find only one phone number format. What about a phone number formatted like 415.555.4242 or (415) 555-4242? And what if the phone number had an extension, like 415-555-4242 x99? The `is_phone_number()` function would fail to find them. You could add yet more code for these additional patterns, but there is an easier way to tackle the problem.

Regular expressions, called *regexes* for short, are a sort of mini language that describes a pattern of text. For example, the characters `\d` in a regex stand for a decimal numeral between 0 and 9. Python uses the regex string `r'\d\d\d-\d\d\d-\d\d\d\d'` to match the same text pattern the previous `is_phone_number()` function did: a string of three numbers, a hyphen, three more numbers, another hyphen, and four numbers. Any other string would not match the `r'\d\d\d-\d\d\d-\d\d\d\d'` regex.

Regular expressions can be much more sophisticated than this one. For example, adding a numeral, such as 3, in curly brackets (`{3}`) after a pattern is like saying, "Match this pattern three times." So the slightly shorter regex `r'\d{3}-\d{3}-\d{4}'` also matches the phone number pattern.

Note that we often write regex strings as raw strings, with the `r` prefix. This is useful, as regex strings often have backslashes. Without using raw strings, we would have to enter expressions such as`'\\d'`.

Before we cover all of the details of regular expression syntax, let's go over how to use them in Python. We'll stick with the example regular expression string `r'\d{3}-\d{3}-\d{4}'` used to find US phone numbers in a text string `'My number is 415-555-4242'`. The general process of using regular expressions in Python involves four steps:

1. Import the `re` module.
2. Pass the regex string to `re.compile()` to get a `Pattern` object.
3. Pass the text string to the `Pattern` object's `search()` method to get a `Match` object.

4. Call the Match object's group() method to get the string of the matched text.

In the interactive shell, these steps look like this:

```
>>> import re
>>> phone_num_pattern_obj = re.compile(r'\d{3}-\d{3}-\d{4}')
>>> match_obj = phone_num_pattern_obj.search('My number is 415-555-4242.')
>>> match_obj.group()
'415-555-4242'
```

All regex functions in Python are in the re module. Most of the examples in this chapter will require the re module, so remember to import it at the beginning of the program. Otherwise, you'll get a `NameError: name 're' is not defined` error message. As with importing any module, you need to import it only once per program or interactive shell session.

Passing the regular expression string to re.compile() returns a Pattern object. You only need to compile the Pattern object once; after that, you can call the Pattern object's search() method for as many different text strings as you want.

A Pattern object's search() method searches the string it is passed for any matches to the regex. The search() method will return None if the regex pattern isn't found in the string. If the pattern *is* found, the search() method returns a Match object, which will have a group() method that returns a string of the matched text.

NOTE *While I encourage you to enter the example code into the interactive shell, you could also make use of web-based regular expression testers, which can show you exactly how a regex matches a piece of text that you enter. I recommend the testers at* https://pythex.org *and* https://regex101.com. *Different programming languages have slightly different regular expression syntax, so be sure to select the "Python" flavor on these websites.*

The Syntax of Regular Expressions

Now that you know the basic steps for creating and finding regular expression objects using Python, you're ready to learn the full range of regular expression syntax. In this section, you'll learn how to group regular expression elements together with parentheses, escape special characters, match several alternative groups with the pipe character, and return all matches with the findall() method.

Grouping with Parentheses

Say you want to separate one smaller part of the matched text, such as the area code, from the rest of the phone number (to, for example, perform some operation on it). Adding parentheses will create *groups* in the regex string: `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`. Then, you can use the `group()` method of `Match` objects to grab the matching text from just one group.

The first set of parentheses in a regex string will be group `1`. The second set will be group `2`. By passing the integer `1` or `2` to the `group()` method, you can grab different parts of the matched text. Passing `0` or nothing to the `group()` method will return the entire matched text. Enter the following into the interactive shell:

```
>>> import re
>>> phone_re = re.compile(r'(\d\d\d)-(\d\d\d-\d\d\d\d)')
>>> mo = phone_re.search('My number is 415-555-4242.')
>>> mo.group(1)  # Returns the first group of the matched text
'415'
>>> mo.group(2)  # Returns the second group of the matched text
'555-4242'
>>> mo.group(0)  # Returns the full matched text
'415-555-4242'
>>> mo.group()  # Also returns the full matched text
'415-555-4242'
```

If you would like to retrieve all the groups at once, use the `groups()` method (note the plural form in the name):

```
>>> mo.groups()
('415', '555-4242')
>>> area_code, main_number = mo.groups()
>>> print(area_code)
415
>>> print(main_number)
555-4242
```

Because `mo.groups()` returns a tuple of multiple values, you can use the multiple-assignment trick to assign each value to a separate variable, as in the previous `area_code, main_number = mo.groups()` line.

Using Escape Characters

Parentheses create groups in regular expressions and are not interpreted as part of the text pattern. So, what do you do if you need to match a parenthesis in your text? For instance, maybe the phone numbers you are trying to match have the area code set in parentheses: `'(415) 555-4242'`.

In this case, you need to escape the `(` and `)` characters with a backslash. The `\(` and `\)` escaped parentheses will be interpreted as part of the pattern you are matching. Enter the following into the interactive shell:

```
>>> pattern = re.compile(r'(\(\d\d\d\)) (\d\d\d-\d\d\d\d)')
>>> mo = pattern.search('My phone number is (415) 555-4242.')
>>> mo.group(1)
'(415)'
>>> mo.group(2)
'555-4242'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The `\(` and `\)` escape characters in the raw string passed to `re.compile()` will match actual parenthesis characters. In regular expressions, the following characters have special meanings:

```
# $ & ( ) * + - . ? [ \ ] ^ { | } ~
```

If you want to detect these characters as part of your text pattern, you need to escape them with a backslash:

```
\# \$ \& \( \) \* \+ \- \. \? \[ \\ \] \^ \{ \| \} \~
```

Always double-check that you haven't mistaken escaped parentheses `\(` and `\)` for unescaped parentheses `(` and `)` in a regular expression. If you receive an error message about "missing)" or "unbalanced parenthesis," you may have forgotten to include the closing unescaped parenthesis for a group, like in this example:

```
>>> import re
>>> re.compile(r'(\(Parentheses\)')
Traceback (most recent call last):
--snip--
re.error: missing ), unterminated subpattern at position 0
```

The error message tells you that there is an opening parenthesis at index `0` of the `r'(\(Parentheses\)'` string that is missing its corresponding closing parenthesis. Using the Humre module described later in this chapter helps prevent these kinds of typos.

Matching Characters from Alternate Groups

The | character is called a *pipe*, and it's used as the *alternation operator* in regular expressions. You can use it anywhere you want to match one of multiple expressions. For example, the regular expression `r'Cat|Dog'` will match either `'Cat'` or `'Dog'`.

You can also use the pipe to match one of several patterns as part of your regex. For example, say you wanted to match any of the strings `'Caterpillar'`, `'Catastrophe'`, `'Catch'`, or `'Category'`. Since all of these strings start with `Cat`, it would be nice if you could specify that prefix only once. You can do this by using the pipe within parentheses to separate the possible suffixes. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'Cat(erpillar|astrophe|ch|egory)')
>>> match = pattern.search('Catch me if you can.')
>>> match.group()
'Catch'
>>> match.group(1)
'ch'
```

The method call `match.group()` returns the full matched text `'Catch'`, while `match.group(1)` returns just the part of the matched text inside the first parentheses group, `'ch'`. By using the pipe character and grouping parentheses, you can specify several alternative patterns you would like your regex to match.

If you need to match an actual pipe character, escape it with a backslash, like `\|`.

Returning All Matches

In addition to a `search()` method, `Pattern` objects have a `findall()` method. While `search()` will return a `Match` object of the *first* matched text in the searched string, the `findall()` method will return the strings of *every* match in the searched string.

There is one detail you need to keep in mind when using `findall()`. The method returns a list of strings *as long as there are no groups in the regular expression*. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'\d{3}-\d{3}-\d{4}')  # This regex has no groups.
>>> pattern.findall('Cell: 415-555-9999 Work: 212-555-0000')
['415-555-9999', '212-555-0000']
```

If there *are* groups in the regular expression, then `findall()` will return a list of tuples. Each tuple represents a single match, and the tuple has strings for each group in the regex. To see this behavior in action, enter the following into the interactive shell (and notice that the regular expression being compiled now has groups in parentheses):

```
>>> import re
>>> pattern = re.compile(r'(\d{3})-(\d{3})-(\d{4})')  # This regex has groups.
>>> pattern.findall('Cell: 415-555-9999 Work: 212-555-0000')
[('415', '555', '9999'), ('212', '555', '0000')]
```

Also keep in mind that `findall()` doesn't overlap matches. For example, matching three numbers with the regex string `r'\d{3}'` matches the first three numbers in `'1234'` but not the last three:

```
>>> import re
>>> pattern = re.compile(r'\d{3}')
>>> pattern.findall('1234')
['123']
>>> pattern.findall('12345')
['123']
>>> pattern.findall('123456')
['123', '456']
```

Because the first three digits in `'1234'` have been matched as `'123'`, the digits `'234'` won't be included in further matches, even though they fit the `r'\d{3}'` pattern.

Qualifier Syntax: What Characters to Match

Regular expressions are split into two parts: the *qualifiers* that dictate what characters you are trying to match followed by the *quantifiers* that dictate how many characters you are trying to match. In the `r'\d{3}-\d{3}-\d{4}'` phone number regex string example we've been using, the `r'\d'` and `'-'` parts are qualifiers and the `'{3}'` and `'{4}'` are quantifiers. Let's now examine the syntax of qualifiers.

Using Character Classes and Negative Character Classes

Although you can define a single character to match, as we've done in the previous examples, you can also define a set of characters to match inside square brackets. This set is called a *character class*. For example, the character class `[aeiouAEIOU]` will match any vowel, both lowercase and uppercase. It's the equivalent of writing `a|e|i|o|u|A|E|I|O|U`, but it's easier to type. Enter the following into the interactive shell:

```
>>> import re
>>> vowel_pattern = re.compile(r'[aeiouAEIOU]')
>>> vowel_pattern.findall('RoboCop eats BABY FOOD.')
['o', 'o', 'o', 'e', 'a', 'A', 'O', 'O']
```

You can also include ranges of letters or numbers by using a hyphen. For example, the character class `[a-zA-Z0-9]` will match all lowercase letters, uppercase letters, and numbers.

Note that, inside the square brackets, the normal regular expression symbols are not interpreted as such. This means you do not need to escape characters such as parentheses inside the square brackets if you want to match literal parentheses. For example, the character class `[()]` will match either an open or close parenthesis. You do not need to write this as `[\(\)]`.

By placing a caret character (`^`) just after the character class's opening bracket, you can make a *negative character class*. A negative character class will match all the characters that are *not* in the character class. For example, enter the following into the interactive shell:

```
>>> import re
>>> consonant_pattern = re.compile(r'[^aeiouAEIOU]')
>>> consonant_pattern.findall('RoboCop eats BABY FOOD.')
['R', 'b', 'C', 'p', ' ', 't', 's', ' ', 'B', 'B', 'Y', ' ', 'F', 'D', '.']
```

Now, instead of matching every vowel, we're matching every character that isn't a vowel. Keep in mind that this includes spaces, newlines, punctuation characters, and numbers.

Using Shorthand Character Classes

In the earlier phone number regex example, you learned that `\d` could stand for any numeric digit. That is, `\d` is shorthand for the regular expression 0|1|2|3|4|5|6|7|8|9 or [0-9]. There are many such *shorthand character classes*, as shown in Table 9-1.

Table 9-1: Shorthand Codes for Common Character Classes

Shorthand character class	Represents . . .
\d	Any numeric digit from 0 to 9.
\D	Any character that is *not* a numeric digit from 0 to 9.
\w	Any letter, numeric digit, or the underscore character. (Think of this as matching "word" characters.)
\W	Any character that is *not* a letter, numeric digit, or the underscore character.
\s	Any space, tab, or newline character. (Think of this as matching "space" characters.)
\S	Any character that is *not* a space, tab, or newline character.

Note that while \d matches digits and \w matches digits, letters, and the underscore, there is no shorthand character class that matches only letters. Though you can use the [a-zA-Z] character class, this character class won't match accented letters or non-Roman alphabet letters such as 'é'. Also, remember to use raw strings to escape the backslash: r'\d'.

For example, enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'\d+\s\w+')
>>> pattern.findall('12 drummers, 11 pipers, 10 lords, 9 ladies, 8 maids,
7 swans, 6 geese, 5 rings, 4 birds, 3 hens, 2 doves, 1 partridge')
['12 drummers', '11 pipers', '10 lords', '9 ladies', '8 maids', '7 swans', '
6 geese', '5 rings', '4 birds', '3 hens', '2 doves', '1 partridge']
```

The regular expression \d+\s\w+ will match text that has one or more numeric digits (\d+), followed by a whitespace character (\s), followed by one or more letter/digit/underscore characters (\w+). The findall() method returns all matching strings of the regular expression pattern in a list.

Matching Everything with the Dot Character

The . (or *dot*) character in a regular expression string matches any character except for a newline. For example, enter the following into the interactive shell:

```
>>> import re
>>> at_re = re.compile(r'.at')
>>> at_re.findall('The cat in the hat sat on the flat mat.')
['cat', 'hat', 'sat', 'lat', 'mat']
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Remember that the dot character will match just one character, which is why the text `flat` in the previous example matched only `lat`. To match an actual period, escape the dot with a backslash: `\.`

Being Careful What You Match For

The best and worst thing about regular expressions is that they will match exactly what you ask for. Here are some common points of confusion regarding character classes:

- The `[A-Z]` or `[a-z]` character class matches uppercase or lowercase letters, respectively, but not both. You need to use `[A-Za-z]` to match both cases.
- The `[A-Za-z]` character class matches only plain, unaccented letters. For example, the regex string `r'First Name: ([A-Za-z]+)'` would match “First Name: ” followed by a group of one or more unaccented letters. But singer Sinéad O’Connor’s first name would match up to the *é* only, and the group would be set to `'Sin'`.
- The `\w` character class matches all letters, including accented letters and characters from other alphabets. But it also matches numbers and the underscore character, so the regex string `r'First Name: (\w+)'` may match more than you intended.
- The `\w` character class matches all letters, but the regex string `r'Last Name: (\w+)'` would capture Sinéad O’Connor’s last name only up until the apostrophe character. This means the group would capture her last name as `'O'`.
- Straight and smart quote characters (`'` `"` ‘ ’ “ ”) are considered completely different from each other and must be specified separately.

Real-world data is complicated. Even if your program manages to capture Sinéad O’Connor’s name, it could fail with Jean-Paul Sartre’s name because of the hyphen.

Of course, when software declares a name to be invalid input, it is the software, and not the name, that has a bug; people’s names cannot be invalid. You can learn more about this issue from Patrick McKenzie’s article “Falsehoods Programmers Believe About Names” at *https://www.kalzumeus.com/2010/06/17/falsehoods-programmers-believe-about-names/*. This article spawned a genre of similar “falsehoods programmers believe” pieces about how software mishandles dates, time zones, currencies, postal addresses, genders, airport codes, and love. Watch Carina C. Zona’s 2015 PyCon talk on the topic, “Schemas for the Real World,” at *https://youtu.be/PYYfVqtcWQY*.

Quantifier Syntax: How Many Qualifiers to Match

In a regular expression string, quantifiers follow qualifier characters to dictate how many of them to match. For example, in the phone number regex considered earlier, the `{3}` follows the `\d` to match exactly three digits. If there is no quantifier following a qualifier, the qualifier must appear exactly once: you can think of `r'\d'` as being the same as `r'\d{1}'`.

Matching an Optional Pattern

Sometimes you may want to match a pattern only optionally. That is, the regex should match zero or one of the preceding qualifiers. The `?` character flags the preceding qualifier as optional. For example, enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'42!?')
>>> pattern.search('42!')
<re.Match object; span=(0, 3), match='42!'>
>>> pattern.search('42')
<re.Match object; span=(0, 2), match='42'>
```

The `?` part of the regular expression means that the pattern `!` is optional. So it matches both `42!` (with the exclamation mark) and `42` (without it).

As you're beginning to see, regular expression syntax's reliance on symbols and punctuation makes it tricky to read: the `?` question mark has meaning in regex syntax, but the `!` exclamation mark doesn't. So `r'42!?'` means `'42'` optionally followed by a `'!'`, but `r'42?!'` means `'4'` optionally followed by `'2'` followed by `'!'`:

```
>>> import re
>>> pattern = re.compile(r'42?!')
>>> pattern.search('42!')
<re.Match object; span=(0, 3), match='42!'>
>>> pattern.search('4!')
<re.Match object; span=(0, 2), match='4!'>
>>> pattern.search('42') == None  # No match
True
```

To make multiple characters optional, place them in a group and put the `?` after the group. In the earlier phone number example, you can use `?` to make the regex look for phone numbers that either do or do not have an area code. Enter the following into the interactive shell:

```
>>> pattern = re.compile(r'(\d{3}-)?\d{3}-\d{4}')
>>> match1 = pattern.search('My number is 415-555-4242')
>>> match1.group()
'415-555-4242'

>>> match2 = pattern.search('My number is 555-4242')
>>> match2.group()
'555-4242'
```

You can think of the `?` as saying, "Match zero or one of the group preceding this question mark."

If you need to match an actual question mark character, escape it with `\?`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Matching Zero or More Qualifiers

The * (called the *star* or *asterisk*) means "match zero or more." In other words, the qualifier that precedes the star can occur any number of times in the text. It can be completely absent or repeated over and over again. Take a look at the following example:

```
>>> import re
>>> pattern = re.compile('Eggs( and spam)*')
>>> pattern.search('Eggs')
<re.Match object; span=(0, 4), match='Eggs'>
>>> pattern.search('Eggs and spam')
<re.Match object; span=(0, 13), match='Eggs and spam'>
>>> pattern.search('Eggs and spam and spam')
<re.Match object; span=(0, 22), match='Eggs and spam and spam'>
>>> pattern.search('Eggs and spam and spam and spam')
<re.Match object; span=(0, 31), match='Eggs and spam and spam and spam'>
```

While the `'Eggs'` part of the string must appear once, there can be any number of `' and spam'` following it, including zero instances.

If you need to match an actual star character, prefix the star in the regular expression with a backslash, *.

Matching One or More Qualifiers

While * means "match zero or more," the `+` (or *plus*) means "match one or more." Unlike the star, which does not require its qualifier to appear in the matched string, the plus requires the qualifier preceding it to appear *at least once*. It is not optional. Enter the following into the interactive shell, and compare it with the star regexes in the previous section:

```
>>> pattern = re.compile('Eggs( and spam)+')
>>> pattern.search('Eggs and spam')
<re.Match object; span=(0, 13), match='Eggs and spam'>
>>> pattern.search('Eggs and spam and spam')
<re.Match object; span=(0, 22), match='Eggs and spam and spam'>
>>> pattern.search('Eggs and spam and spam and spam')
<re.Match object; span=(0, 31), match='Eggs and spam and spam and spam'>
```

The regex `'Eggs( and spam)+'` will not match the string `'Eggs'`, because the plus sign requires at least one `' and spam'`.

You'll often use parentheses in your regex strings to group together qualifiers so that a quantifier can apply to the entire group. For example, you could match any combination of dots and dashes of Morse code with `r'(\.|\-)+'` (though this expression would also match invalid Morse code combinations).

If you need to match an actual plus sign character, prefix the plus sign with a backslash to escape it: `\+`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Matching a Specific Number of Qualifiers

If you have a group that you want to repeat a specific number of times, follow the group in your regex with a number in curly brackets. For example, the regex `(Ha){3}` will match the string `'HaHaHa'` but not `'HaHa'`, since the latter has only two repeats of the `(Ha)` group.

Instead of one number, you can specify a range by writing a minimum, a comma, and a maximum in between the curly brackets. For example, the regex `(Ha){3,5}` will match `'HaHaHa'`, `'HaHaHaHa'`, and `'HaHaHaHaHa'`.

You can also leave out the first or second number in the curly brackets to keep the minimum or maximum unbounded. For example, `(Ha){3,}` will match three or more instances of the `(Ha)` group, while `(Ha){,5}` will match zero to five instances. Curly brackets can help make your regular expressions shorter. These two regular expressions match identical patterns:

```
(Ha){3}
HaHaHa
```

So do these two regular expressions:

```
(Ha){3,5}
(HaHaHa)|(HaHaHaHa)|(HaHaHaHaHa)
```

Enter the following into the interactive shell:

```
>>> import re
>>> haRegex = re.compile(r'(Ha){3}')
>>> match1 = haRegex.search('HaHaHa')
>>> match1.group()
'HaHaHa'

>>> match = haRegex.search('HaHa')
>>> match == None
True
```

Here, `(Ha){3}` matches `'HaHaHa'` but not `'Ha'`. Because it doesn't match `'HaHa'`, `search()` returns `None`.

The syntax of the curly bracket quantifier is similar to Python's slice syntax (such as `'Hello, world!'[3:5]`, which evaluates to `'lo'`). But there are key differences. In the regex quantifier, the two numbers are separated by a comma and not a colon. Also, the second number in the quantifier is inclusive: `'(Ha){3,5}'` matches up to *and including* five instances of the `'(Ha)'` qualifier.

Greedy and Non-greedy Matching

Because `(Ha){3,5}` can match three, four, or five instances of `Ha` in the string `'HaHaHaHaHa'`, you may wonder why the `Match` object's call to `group()` in the previous curly bracket example returns `'HaHaHaHaHa'` instead of the shorter

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

possibilities. After all, `'HaHaHa'` and `'HaHaHaHa'` are also valid matches of the regular expression `(Ha){3,5}`.

Python's regular expressions are *greedy* by default, which means that in ambiguous situations, they will match the longest string possible. The *non-greedy* (also called *lazy*) version of the curly brackets, which matches the shortest string possible, must follow the closing curly bracket with a question mark.

Enter the following into the interactive shell, and notice the difference between the greedy and non-greedy forms of the curly brackets searching the same string:

```
>>> import re
>>> greedy_pattern = re.compile(r'(Ha){3,5}')
>>> match1 = greedy_pattern.search('HaHaHaHaHa')
>>> match1.group()
'HaHaHaHaHa'

>>> lazy_pattern = re.compile(r'(Ha){3,5}?')
>>> match2 = lazy_pattern.search('HaHaHaHaHa')
>>> match2.group()
'HaHaHa'
```

Note that the question mark can have two meanings in regular expressions: declaring a lazy match or declaring an optional qualifier. These meanings are entirely unrelated.

It's worth pointing out that, technically, you could get by without using the optional `?` quantifier, or even the `*` and `+` quantifiers:

- The `?` quantifier is the same as `{0,1}`.
- The `*` quantifier is the same as `{0,}`.
- The `+` quantifier is the same as `{1,}`.

However, the `?`, `*`, and `+` quantifiers are common shorthand.

Matching Everything

Sometimes you may want to match everything and anything. For example, say you want to match the string `'First Name:'`, followed by any and all text, followed by `'Last Name:'` and any text once again. You can use the dot-star (`.*`) to stand in for that "anything." Remember that the dot character means "any single character except the newline," and the star character means "zero or more of the preceding character."

Enter the following into the interactive shell:

```
>>> import re
>>> name_pattern = re.compile(r'First Name: (.*) Last Name: (.*)')
>>> name_match = name_pattern.search('First Name: Al Last Name: Sweigart')
>>> name_match.group(1)
'Al'
>>> name_match.group(2)
'Sweigart'
```

The dot-star uses *greedy* mode: it will always try to match as much text as possible. To match any and all text in a *non-greedy* or *lazy* fashion, use the dot, star, and question mark (.*?). As when it's used with curly brackets, the question mark tells Python to match in a non-greedy way.

Enter the following into the interactive shell to see the difference between the greedy and non-greedy expressions:

```
>>> import re
>>> lazy_pattern = re.compile(r'<.*?>')
>>> match1 = lazy_pattern.search('<To serve man> for dinner.>')
>>> match1.group()
'<To serve man>'

>>> greedy_re = re.compile(r'<.*>')
>>> match2 = greedy_re.search('<To serve man> for dinner.>')
>>> match2.group()
'<To serve man> for dinner.>'
```

Both regexes roughly translate to "Match an opening angle bracket, followed by anything, followed by a closing angle bracket." But the string `'<To serve man> for dinner.>'` has two possible matches for the closing angle bracket. In the non-greedy version of the regex, Python matches the shortest possible string: `'<To serve man>'`. In the greedy version, Python matches the longest possible string: `'<To serve man> for dinner.>'`.

Matching Newline Characters

The dot in `.*` will match everything except a newline. By passing `re.DOTALL` as the second argument to `re.compile()`, you can make the dot character match *all* characters, including the newline character.

Enter the following into the interactive shell:

```
>>> import re
>>> no_newline_re = re.compile('.*')
>>> no_newline_re.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.'

>>> newline_re = re.compile('.*', re.DOTALL)
>>> newline_re.search('Serve the public trust.\nProtect the innocent.
\nUphold the law.').group()
'Serve the public trust.\nProtect the innocent.\nUphold the law.'
```

The regex `no_newline_re`, which did not have `re.DOTALL` passed to the `re.compile()` call that created it, will match everything only up to the first newline character, whereas `newline_re`, which *did* have `re.DOTALL` passed to `re.compile()`, matches everything. This is why the `newline_re.search()` call matches the full string, including its newline characters.

Matching at the Start and End of a String

You can use the caret symbol (^) at the start of a regex to indicate that a match must occur at the *beginning* of the searched text. Likewise, you can put a dollar sign ($) at the end of the regex to indicate that the string must *end* with this regex pattern. And you can use the ^ and $ together to indicate that the entire string must match the regex—that is, it's not enough for a match to be made on some subset of the string.

For example, the r'^Hello' regular expression string matches strings that begin with 'Hello'. Enter the following into the interactive shell:

```
>>> import re
>>> begins_with_hello = re.compile(r'^Hello')
>>> begins_with_hello.search('Hello, world!')
<re.Match object; span=(0, 5), match='Hello'>
>>> begins_with_hello.search('He said "Hello."') == None
True
```

The r'\d$' regular expression string matches strings that end with a numeric character between 0 and 9. Enter the following into the interactive shell:

```
>>> import re
>>> ends_with_number = re.compile(r'\d$')
>>> ends_with_number.search('Your number is 42')
<re.Match object; span=(16, 17), match='2'>
>>> ends_with_number.search('Your number is forty two.') == None
True
```

The r'^\d+$' regular expression string matches strings that both begin and end with one or more numeric characters. Enter the following into the interactive shell:

```
>>> import re
>>> whole_string_is_num = re.compile(r'^\d+$')
>>> whole_string_is_num.search('1234567890')
<re.Match object; span=(0, 10), match='1234567890'>
>>> whole_string_is_num.search('12345xyz67890') == None
True
```

The last two search() calls in the previous interactive shell example demonstrate how the entire string must match the regex if ^ and $ are used. (I always confuse the meanings of these two symbols, so I use the mnemonic "carrots cost dollars" to remind myself that the caret comes first and the dollar sign comes last.)

You can also use \b to make a regex pattern match only on a *word boundary*: the start of a word, end of a word, or both the start and end of a word. In this case, a "word" is a sequence of letters separated by non-letter

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

characters. For example, `r'\bcat.*?\b'` matches a word that begins with `'cat'` followed by any other characters up to the next word boundary:

```
>>> import re
>>> pattern = re.compile(r'\bcat.*?\b')
>>> pattern.findall('The cat found a catapult catalog in the catacombs.')
['cat', 'catapult', 'catalog', 'catacombs']
```

The `\B` syntax matches anything that is not a word boundary:

```
>>> import re
>>> pattern = re.compile(r'\Bcat\B')
>>> pattern.findall('certificate')  # Match
['cat']
>>> pattern.findall('catastrophe')  # No match
[]
```

It is useful for finding matches in the middle of a word.

A REVIEW OF REGEX SYMBOLS

This chapter has covered a lot of notation so far, so here's a quick review of what you've learned about basic regular expression syntax:

- The `?` matches zero or one instance of the preceding qualifier.
- The `*` matches zero or more instances of the preceding qualifier.
- The `+` matches one or more instances of the preceding qualifier.
- The `{n}` matches exactly *n* instances of the preceding qualifier.
- The `{n,}` matches *n* or more instances of the preceding qualifier.
- The `{,m}` matches 0 to *m* instances of the preceding qualifier.
- The `{n,m}` matches at least *n* and at most *m* instances of the preceding qualifier.
- `{n,m}?` or `*?` or `+?` performs a non-greedy match of the preceding qualifier.
- `^spam` means the string must begin with *spam*.
- `spam$` means the string must end with *spam*.
- The `.` matches any character, except newline characters.
- The `\d`, `\w`, and `\s` match a digit, word, or space character, respectively.
- The `\D`, `\W`, and `\S` match anything except a digit, word, or space character, respectively. `[abc]` matches any character between the square brackets (such as *a*, *b*, or *c*).
- `[^abc]` matches any character that isn't between the square brackets.
- `(Hello)` groups `'Hello'` together as a single qualifier.

Case-insensitive Matching

Normally, regular expressions match text with the exact casing you specify. For example, the following regexes match completely different strings:

```
>>> import re
>>> pattern1 = re.compile('RoboCop')
>>> pattern2 = re.compile('ROBOCOP')
>>> pattern3 = re.compile('robOcop')
>>> pattern4 = re.compile('RobocOp')
```

But sometimes you care only about matching the letters, and aren't worried about whether they're uppercase or lowercase. To make your regex case-insensitive, you can pass `re.IGNORECASE` or `re.I` as a second argument to `re.compile()`. Enter the following into the interactive shell:

```
>>> import re
>>> pattern = re.compile(r'robocop', re.I)
>>> pattern.search('RoboCop is part man, part machine, all cop.').group()
'RoboCop'

>>> pattern.search('ROBOCOP protects the innocent.').group()
'ROBOCOP'

>>> pattern.search('Have you seen robocop?').group()
'robocop'
```

The regular expression now matches strings with any casing.

Substituting Strings

Regular expressions don't merely find text patterns; they can also substitute new text in place of those patterns. The `sub()` method for `Pattern` objects accepts two arguments. The first is a string that should replace any matches. The second is the string of the regular expression. The `sub()` method returns a string with the substitutions applied.

For example, enter the following into the interactive shell to replace secret agents' names with `CENSORED`:

```
>>> import re
>>> agent_pattern = re.compile(r'Agent \w+')
>>> agent_pattern.sub('CENSORED', 'Agent Alice contacted Agent Bob.')
'CENSORED contacted CENSORED.'
```

Sometimes you may need to use the matched text itself as part of the substitution. In the first argument to `sub()`, you can include `\1`, `\2`, `\3`, and so on, to mean "Enter the text of group `1`, `2`, `3`, and so on, in the substitution." This syntax is called a *back reference*.

For example, say you want to censor the names of the secret agents by showing just the first letters of their names. To do this, you could use the regex `Agent (\w)\w*` and pass `r'\1****'` as the first argument to `sub()`:

```
>>> import re
>>> agent_pattern = re.compile(r'Agent (\w)\w*')
>>> agent_pattern.sub(r'\1****', 'Agent Alice contacted Agent Bob.')
'A**** contacted B****.'
```

The `\1` in the regular expression string is replaced by whatever text was matched by group `1`—that is, the `(\w)` group of the regular expression.

Managing Complex Regexes with Verbose Mode

Regular expressions are fine if the text pattern you need to match is simple. But matching complicated text patterns might require long, convoluted regular expressions. You can mitigate this complexity by telling the `re.compile()` function to ignore whitespace and comments inside the regular expression string. Enable this "verbose mode" by passing the variable `re.VERBOSE` as the second argument to `re.compile()`.

Now, instead of a hard-to-read regular expression like this

```
pattern = re.compile(r'((\d{3}|\(\d{3}\))?(\s|-|\.)?\d{3}(\s|-
|\.)\d{4}(\s*(ext|x|ext\.)\s*\d{2,5})?)')
```

you can spread the regular expression over multiple lines and use comments to label its components, like this:

```
pattern = re.compile(r'''(
    (\d{3}|\(\d{3}\))?  # Area code
    (\s|-|\.)?  # Separator
    \d{3}  # First three digits
    (\s|-|\.)  # Separator
    \d{4}  # Last four digits
    (\s*(ext|x|ext\.)\s*\d{2,5})?  # Extension
    )''', re.VERBOSE)
```

Note how the previous example uses the triple-quote syntax (`'''`) to create a multiline string so that you can spread the regular expression definition over many lines, making it much more legible.

The comment rules inside the regular expression string are the same as for regular Python code: the `#` symbol and everything after it until the end of the line are ignored. Also, the extra spaces inside the multiline string for the regular expression are not considered part of the text pattern to be matched. This lets you organize the regular expression so that it's easier to read.

While verbose mode makes your regex strings more readable, I advise you to instead use the Humre module, covered later in this chapter, to improve the readability of your regular expressions.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Combining re.IGNORECASE, re.DOTALL, and re.VERBOSE

What if you want to use `re.VERBOSE` to write comments in your regular expression, but also want to use `re.IGNORECASE` to ignore capitalization? Unfortunately, the `re.compile()` function takes only a single value as its second argument.

You can get around this limitation by combining the `re.IGNORECASE`, `re.DOTALL`, and `re.VERBOSE` variables using the pipe character (`|`), which in this context is known as the *bitwise or* operator. For example, if you want a regular expression that is case-insensitive *and* includes newlines to match the dot character, you would form your `re.compile()` call like this:

```
>>> some_regex = re.compile('foo', re.IGNORECASE | re.DOTALL)
```

Including all three options in the second argument looks like this:

```
>>> some_regex = re.compile('foo', re.IGNORECASE | re.DOTALL | re.VERBOSE)
```

This syntax is a little old-fashioned and originates from early versions of Python. The details of the bitwise operators are beyond the scope of this book, but check out the resources at *https://nostarch.com/automatestuff3/* for more information. You can also pass other options for the second argument; they're uncommon, but you can read more about them in the resources too.

Project 3: Phone Number and Email Address Extractor

Say you've been given the boring task of finding every phone number and email address in a long web page or document. If you manually scroll through the page, you might end up searching for a long time. But if you had a program that could search the text in your clipboard for phone numbers and email addresses, you could simply press CTRL-A to select all the text, press CTRL-C to copy it to the clipboard, and then run your program. It could replace the text on the clipboard with just the phone numbers and email addresses it finds.

Whenever you're tackling a new project, it can be tempting to dive right into writing code. But more often than not, it's best to take a step back and consider the bigger picture. I recommend first drawing up a high-level plan for what your program needs to do. Don't think about the actual code yet; you can worry about that later. Right now, stick to broad strokes.

For example, your phone number and email address extractor will need to do the following:

- Get the text from the clipboard.
- Find all phone numbers and email addresses in the text.
- Paste them onto the clipboard.

Now you can start thinking about how this might work in code. The code will need to do the following:

- Use the pyperclip module to copy and paste strings.
- Create two regexes, one for matching phone numbers and one for matching email addresses.
- Find all matches (not just the first match) of both regexes.
- Neatly format the matched strings into a single string to paste.
- Display some kind of message if no matches were found in the text.

This list is like a road map for the project. As you write the code, you can focus on each of these steps separately, and each step should seem fairly manageable. They're also expressed in terms of things you already know how to do in Python.

Step 1: Create a Regex for Phone Numbers

First, you have to create a regular expression to search for phone numbers. Create a new file, enter the following, and save it as *phoneAndEmail.py*:

```
import pyperclip, re

phone_re = re.compile(r'''(
    (\d{3}|\(\d{3}\))?  # Area code
    (\s|-|\.)?  # Separator
    (\d{3})  # First three digits
    (\s|-|\.)  # Separator
    (\d{4})  # Last four digits
    (\s*(ext|x|ext\.)\s*(\d{2,5}))?  # Extension
    )''', re.VERBOSE)

# TODO: Create email regex.

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The TODO comments are just a skeleton for the program. They'll be replaced as you write the actual code.

The phone number begins with an *optional* area code, so we follow the area code group with a question mark. Since the area code can be just three digits (that is, \d{3}) *or* three digits within parentheses (that is, \(\d{3}\)), you should have a pipe joining those parts. You can add the regex comment # Area code to this part of the multiline string to help you remember what (\d{3}|\(\d{3}\))? is supposed to match.

The phone number separator character can be an *optional* space (\s), hyphen (-), or period (.), so we should also join these parts using pipes. The next few parts of the regular expression are straightforward: three digits, followed by another separator, followed by four digits. The last part is an

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

optional extension made up of any number of spaces followed by `ext`, `x`, or `ext.`, followed by two to five digits.

NOTE *It's easy to get mixed up when writing regular expressions that contain groups with parentheses* `( )` *and escaped parentheses* `\( \)`*. Remember to double-check that you're using the correct syntax if you get a "missing), unterminated subpattern" error message.*

Step 2: Create a Regex for Email Addresses

You will also need a regular expression that can match email addresses. Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--

# Create email regex.
email_re = re.compile(r'''(
  ❶ [a-zA-Z0-9._%+-]+  # username
  ❷ @  # @ symbol
  ❸ [a-zA-Z0-9.-]+  # domain name
    (\.[a-zA-Z]{2,4})  # dot-something
    )''', re.VERBOSE)

# TODO: Find matches in clipboard text.

# TODO: Copy results to the clipboard.
```

The username part of the email address ❶ consists of one or more characters that can be any of the following: lowercase and uppercase letters, numbers, a dot, an underscore, a percent sign, a plus sign, or a hyphen. You can put all of these into a character class: `[a-zA-Z0-9._%+-]`.

The domain and username are separated by an `@` symbol ❷. The domain name ❸ has a slightly less permissive character class, with only letters, numbers, periods, and hyphens: `[a-zA-Z0-9.-]`. Last is the "dot-com" part (technically known as the *top-level domain*), which can really be dot-anything.

The format for email addresses has a lot of weird rules. This regular expression won't match every possible valid email address, but it will match almost any typical email address you'll encounter.

Step 3: Find All Matches in the Clipboard Text

Now that you've specified the regular expressions for phone numbers and email addresses, you can let Python's `re` module do the hard work of finding all the matches on the clipboard. The `pyperclip.paste()` function will get a string value of the text on the clipboard, and the `findall()` regex method will return a list of tuples.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--

# Find matches in clipboard text.
text = str(pyperclip.paste())

❶ matches = []
❷ for groups in phone_re.findall(text):
    phone_num = '-'.join([groups[1], groups[3], groups[5]])
    if groups[6] != '':
        phone_num += ' x' + groups[6]
    matches.append(phone_num)
❸ for groups in email_re.findall(text):
    matches.append(groups)

# TODO: Copy results to the clipboard.
```

There is one tuple for each match, and each tuple contains strings for each group in the regular expression. Remember that group `0` matches the entire regular expression, so the group at index `0` of the tuple is the one you are interested in.

As you can see at ❶, you'll store the matches in a list variable named `matches`. It starts off as an empty list and a couple of `for` loops. For the email addresses, you append group `0` of each match ❸. For the matched phone numbers, you don't want to just append group `0`. While the program *detects* phone numbers in several formats, you want the phone number appended to be in a single, standard format. The `phone_num` variable contains a string built from groups `1`, `3`, `5`, and `8` of the matched text ❷. (These groups are the area code, first three digits, last four digits, and extension.)

Step 4: Join the Matches into a String

Now that you have the email addresses and phone numbers as a list of strings in `matches`, you want to put them on the clipboard. The `pyperclip.copy()` function takes only a single string value, not a list of strings, so you must call the `join()` method on `matches`.

Make your program look like the following:

```
import pyperclip, re

phone_re = re.compile(r'''(
--snip--
for groups in email_re.findall(text):
    matches.append(groups[0])

# Copy results to the clipboard.
```

```
if len(matches) > 0:
    pyperclip.copy('\n'.join(matches))
    print('Copied to clipboard:')
    print('\n'.join(matches))
else:
    print('No phone numbers or email addresses found.')
```

To make it easier to see that the program is working, we also print any matches you find to the terminal window. If no phone numbers or email addresses were found, the program tells the user this.

To test your program, open your web browser to the No Starch Press contact page at *https://nostarch.com/contactus/*, press CTRL-A to select all the text on the page, and press CTRL-C to copy it to the clipboard. When you run this program, the output should look something like this:

```
Copied to clipboard:
800-555-7240
415-555-9900
415-555-9950
info@nostarch.com
media@nostarch.com
academic@nostarch.com
info@nostarch.com
```

You can modify this script to search for mailing addresses, social media handles, and many other types of text patterns.

Ideas for Similar Programs

Identifying patterns of text (and possibly substituting them with the `sub()` method) has many different potential applications. For example, you could do the following:

- Find website URLs that begin with *http://* or *https://*.
- Clean up dates in different date formats (such as 3/14/2030, 03-14-2030, and 2030/3/14) by replacing them with dates in a single, standard format.
- Remove sensitive information such as Social Security numbers or credit card numbers.
- Find common typos, such as multiple spaces between words, accidentally accidentally repeated words, or multiple exclamation marks at the ends of sentences. Those are annoying!!

Humre: A Module for Human-Readable Regexes

Code is read far more often than it's written, so it's important for your code to be readable. But the punctuation-dense syntax of regular expressions can be hard for even experienced programmers to read. To solve this, the third-party Humre Python module takes the good ideas of verbose mode even further by using human-readable, plain-English names to create readable regex code. You can install Humre by following the instructions in Appendix A.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Let's go back to the r'\d{3}-\d{3}-\d{4}' phone number example from the beginning of this chapter. The functions and constants in Humre can produce the same regex string with plain English:

```
>>> from humre import *
>>> phone_regex = exactly(3, DIGIT) + '-' + exactly(3, DIGIT) + '-' + exactly(4, DIGIT)
>>> phone_regex
'\\d{3}-\\d{3}-\\d{4}'
```

Humre's constants (like DIGIT) contain strings, and Humre's functions (like exactly()) return strings. Humre doesn't replace the re module. Rather, it produces regex strings that can be passed to re.compile():

```
>>> import re
>>> pattern = re.compile(phone_regex)
>>> pattern.search('My number is 415-555-4242')
<re.Match object; span=(13, 25), match='415-555-4242'>
```

Humre has constants and functions for each feature of regular expression syntax. You can then concatenate the constants and returned strings like any other string. For example, here are Humre's constants for the shorthand character classes:

- DIGIT and NONDIGIT represent r'\d' and r'\D', respectively.
- WORD and NONWORD represent r'\w' and r'\W', respectively.
- WHITESPACE and NONWHITESPACE represent r'\s' and r'\S', respectively.

A common source of regex bugs is forgetting which characters need to be escaped. You can use Humre's constants instead of typing the escaped character yourself. For example, say you want to match a single-digit floating-point number with one digit after the decimal point, like '0.9' or '4.5'. However, if you use the regex string r'\d.\d', you might not realize that the dot matches a period (as in '4.5') but also matches any other character (as in '4A5').

Instead, use Humre's PERIOD constant, which contains the string r'\.'. The expression DIGIT + PERIOD + DIGIT evaluates to r'\d\.\d' and makes it much more obvious what the regex intends to match.

The following Humre constants exist for escaped characters:

```
PERIOD           OPEN_PAREN       OPEN_BRACKET     PIPE
DOLLAR_SIGN      CLOSE_PAREN      CLOSE_BRACKET    CARET
QUESTION_MARK    ASTERISK         OPEN_BRACE       TILDE
HASHTAG          PLUS             CLOSE_BRACE
AMPERSAND        MINUS            BACKSLASH
```

There are also constants for NEWLINE, TAB, QUOTE, and DOUBLE_QUOTE. Back references from r'\1' to r'\99' are represented as BACK_1 to BACK_99.

However, you'll make the largest readability gains by using Humre's functions. Table 9-2 shows these functions and their equivalent regular expression syntax.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Table 9-2: Humre Functions

Humre function	Regex string
`group('A')`	`r'(A)'`
`optional('A')`	`r'A?'`
`either('A', 'B', 'C')`	`r'A|B|C'`
`exactly(3, 'A')`	`'A{3}'`
`between(3, 5, 'A')`	`'A{3,5}'`
`at_least(3, 'A')`	`'A{3,}'`
`at_most(3, 'A')`	`'A{,3}'`
`chars('A-Z')`	`'[A-Z]'`
`nonchars('A-Z')`	`'[^A-Z]'`
`zero_or_more('A')`	`'A*'`
`zero_or_more_lazy('A')`	`'A*?'`
`one_or_more('A')`	`'A+'`
`one_or_more_lazy('A')`	`'A+?'`
`starts_with('A')`	`'^A'`
`ends_with('A')`	`'A$'`
`starts_and_ends_with('A')`	`'^A$'`
`named_group('name', 'A')`	`'(?P<name>A)'`

Humre also has several convenience functions that combine common pairs of function calls. For example, instead of using `optional(group('A'))` to create `'(A)?'`, you can simply call `optional_group('A')`. Table 9-3 has the full list of Humre convenience functions.

Table 9-3: Humre Convenience Functions

Convenience function	Function equivalent	Regex string
`optional_group('A')`	`optional(group('A'))`	`'(A)?'`
`group_either('A')`	`group(either('A', 'B', 'C'))`	`'(A|B|C)'`
`exactly_group(3, 'A')`	`exactly(3, group('A'))`	`'(A){3}'`
`between_group(3, 5, 'A')`	`between(3, 5, group('A'))`	`'(A){3,5}'`
`at_least_group (3, 'A')`	`at_least(3, group('A'))`	`'(A){3,}'`
`at_most_group (3, 'A')`	`at_most(3, group('A'))`	`'(A){,3}'`
`zero_or_more_group('A')`	`zero_or_more(group('A'))`	`'(A)*'`
`zero_or_more_lazy_group('A')`	`zero_or_more_lazy(group('A'))`	`'(A)*?'`
`one_or_more_group('A')`	`one_or_more(group('A'))`	`'(A)+'`
`one_or_more_lazy_group('A')`	`one_or_more_lazy(group('A'))`	`'(A)+?'`

All of Humre's functions except either() and group_either() allow you to pass multiple strings to automatically join them. This means that calling group(DIGIT, PERIOD, DIGIT) produces the same regex string as group(DIGIT + PERIOD + DIGIT). They both return the regex string r'(\d\.\d)'.

Finally, Humre has constants for common regex patterns:

ANY_SINGLE The . pattern that matches any single character (except newlines)

ANYTHING_LAZY The lazy .*? zero or more pattern

ANYTHING_GREEDY The greedy .* zero or more pattern

SOMETHING_LAZY The lazy .+? one or more pattern

SOMETHING_GREEDY The greedy .+ one or more pattern

The readability of regex written with Humre becomes more obvious when you consider large, complicated regular expressions. Let's rewrite the phone number regex from the previous phone number extractor project using Humre:

```
import re
from humre import *
phone_regex = group(
  optional_group(either(exactly(3, DIGIT),  # Area code
                      OPEN_PAREN + exactly(3, DIGIT) + CLOSE_PAREN)),
  optional(group_either(WHITESPACE, '-', PERIOD)),  # Separator
  group(exactly(3, DIGIT)),  # First three digits
  group_either(WHITESPACE, '-', PERIOD),  # Separator
  group(exactly(4, DIGIT)),  # Last four digits
  optional_group(  # Extension
    zero_or_more(WHITESPACE),
    group_either('ext', 'x', r'ext\.'),
    zero_or_more(WHITESPACE),
    group(between(2, 5, DIGIT))
    )
  )

pattern = re.compile(phone_regex)
match = pattern.search('My number is 415-555-1212.')
print(match.group())
```

When you run this program, the output is this:

```
415-555-1212
```

This code is much more verbose than even the verbose mode regex. It helps to import Humre using the from humre import * syntax so that you don't need to put humre. before every function and constant. But the length of the code doesn't matter as much as the readability.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You can switch your existing regular expressions to Humre code by calling the `humre.parse()` function, which returns a string of Python source code:

```
>>> import humre
>>> humre.parse(r'\d{3}-\d{3}-\d{4}')
"exactly(3, DIGIT) + '-' + exactly(3, DIGIT) + '-' + exactly(4, DIGIT)"
```

When combined with a modern editor such as PyCharm or Visual Studio Code, Humre offers several further advantages:

- You can indent your code to make it obvious which parts of the regex contain which other parts.
- Your editor's parentheses matching works.
- Your editor's syntax highlighting works.
- Your editor's linter and type hints tool picks up typos.
- Your editor's autocomplete fills in the function and constant names.
- Humre handles raw strings and escaping for you.
- You can put Python comments alongside your Humre code.
- Typos cause more helpful error messages.

Many experienced programmers will object to using anything other than the standard, complicated, unreadable regular expression syntax. As programmer Peter Bhat Harkins once said, "One of the most irritating things programmers do regularly is feel so good about learning a hard thing that they don't look for ways to make it easy, or even oppose things that would do so."

However, if a co-worker objects to your use of Humre, you can simply print the underlying regex string that your Humre code generates and put it back into your source code. For example, the contents of the `phone_regex` variable in the phone number extractor project are as follows:

```
r'((\d{3}|\(\d{3}\))?(\s|-|\.)?(\d{3})(\s|-|\.)(\d{4})(\s*(ext|x|ext\.)\s*(\d{2,5}))?)'
```

Your co-worker is welcome to use this regular expression string if they feel it is more appropriate.

Summary

While a computer can search for text quickly, it must be told precisely what to look for. Regular expressions allow you to specify the pattern of characters you are looking for, rather than the exact text itself. In fact, some word processing and spreadsheet applications provide find-and-replace features that allow you to search using regular expressions. The punctuation-heavy syntax of regular expressions is composed of qualifiers that detail what to match and quantifiers that detail how many to match.

The re module that comes with Python lets you compile a regex string into a `Pattern` object. These objects have several methods: `search()`, to find

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

a single match; `findall()`, to find all matching instances; and `sub()`, to do a find-and-replace substitution of text.

You can find out more in the official Python documentation at *https://docs.python.org/3/library/re.html*. Another useful resource is the tutorial website *https://www.regular-expressions.info*. The Humre page on the Python Package Index is *https://pypi.org/project/Humre/*.

Practice Questions

1. What is the function that returns `Regex` objects?
2. Why are raw strings often used when creating `Regex` objects?
3. What does the `search()` method return?
4. How do you get the actual strings that match the pattern from a `Match` object?
5. In the regex created from `r'(\d\d\d)-(\d\d\d-\d\d\d\d)'`, what does group `0` cover? Group `1`? Group `2`?
6. Parentheses and periods have specific meanings in regular expression syntax. How would you specify that you want a regex to match actual parentheses and period characters?
7. The `findall()` method returns a list of strings or a list of tuples of strings. What makes it return one or the other?
8. What does the `|` character signify in regular expressions?
9. What two things does the `?` character signify in regular expressions?
10. What is the difference between the `+` and `*` characters in regular expressions?
11. What is the difference between `{3}` and `{3,5}` in regular expressions?
12. What do the `\d`, `\w`, and `\s` shorthand character classes signify in regular expressions?
13. What do the `\D`, `\W`, and `\S` shorthand character classes signify in regular expressions?
14. What is the difference between the `.*` and `.*?` regular expressions?
15. What is the character class syntax to match all numbers and lowercase letters?
16. How do you make a regular expression case-insensitive?
17. What does the `.` character normally match? What does it match if `re.DOTALL` is passed as the second argument to `re.compile()`?
18. If `num_re = re.compile(r'\d+')`, what will `num_re.sub('X', '12 drummers, 11 pipers, five rings, 3 hens')` return?
19. What does passing `re.VERBOSE` as the second argument to `re.compile()` allow you to do?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Practice Programs

For practice, write programs to do the following tasks.

Strong Password Detection

Write a function that uses regular expressions to make sure the password string it is passed is strong. A strong password has several rules: it must be at least eight characters long, contain both uppercase and lowercase characters, and have at least one digit. Hint: It's easier to test the string against multiple regex patterns than to try to come up with a single regex that can validate all the rules.

Regex Version of the strip() Method

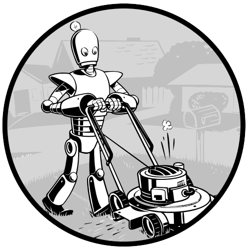
Write a function that takes a string and does the same thing as the `strip()` string method. If no other arguments are passed other than the string to strip, then the function should remove whitespace characters from the beginning and end of the string. Otherwise, the function should remove the characters specified in the second argument to the function.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

10

READING AND WRITING FILES



Variables are a fine way to store data while your program is running, but if you want your data to persist even after your program has finished, you need to save it to a file. You can think of a file's contents as a single string value, potentially gigabytes in size. In this chapter, you'll learn how to use Python to create, read, and save files on the hard drive.

Files and Filepaths

A file has two key properties: a *filename* (usually written as one word) and a *path*. The path specifies the location of a file on the computer. For example, there is a file on my Windows laptop with the filename *project.docx* in the path *C:\Users\Al\Documents*. The part of the filename after the last period is called the file's *extension* and tells you a file's type. The filename *project.docx* is a Word

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

document, and *Users*, *Al*, and *Documents* all refer to *folders* (also called *directories*). Folders can contain files and other folders (called *subfolders*). For example, *project.docx* is in the *Documents* folder, which is inside the *Al* folder, which is inside the *Users* folder. Figure 10-1 shows this folder organization.

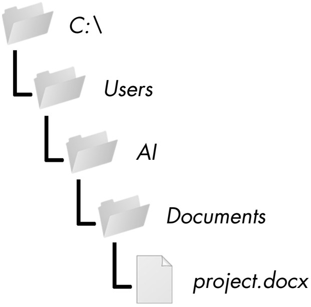


Figure 10-1: A file in a hierarchy of folders

The *C:\* part of the path is the *root folder*, which contains all the other folders. On Windows, the root folder is named *C:\* and is also called the *C: drive*. On macOS and Linux, the root folder is */*. In this book, I'll use the Windows-style root folder, *C:\*. If you are entering the interactive shell examples on macOS or Linux, enter / instead.

Additional *volumes*, such as a DVD drive or USB flash drive, will appear differently on different operating systems. On Windows, they appear as new, lettered root drives, such as *D:\* or *E:\*. On macOS, they appear as new folders under the */Volumes* folder. On Linux, they appear as new folders under the */mnt* ("mount") folder. Also note that while folder names and filenames are not case-sensitive on Windows and macOS, they are case-sensitive on Linux.

NOTE *Because your system probably has different files and folders on it than mine, you won't be able to follow every example in this chapter exactly. Still, try to follow along using folders that exist on your computer.*

Standardizing Path Separators

On Windows, paths are written using backslashes (\) as the separator between folder names. The macOS and Linux operating systems, however, use the forward slash (/) as their path separator.

The `Path()` function in the `pathlib` module handles all operating systems, so the best practice is to use forward slashes in your Python code. If you pass it the string values of individual file and folder names in your path, `Path()` will return a string with a filepath using the correct path separators. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('spam', 'bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> str(Path('spam', 'bacon', 'eggs'))
'spam\\bacon\\eggs'
```

While the `WindowsPath` object may use / forward slashes, converting it to a string with the `str()` function requires using \ backslashes. Note that the convention for importing `pathlib` is to run `from pathlib import Path`, since otherwise we'd have to enter `pathlib.Path` everywhere `Path` shows up in our code. Not only is this extra typing redundant, but it's also redundant.

I'm running this chapter's interactive shell examples on Windows, so `Path('spam', 'bacon', 'eggs')` returned a `WindowsPath` object for the joined path, represented as `WindowsPath('spam/bacon/eggs')`. Even though Windows uses backslashes, the `WindowsPath` representation in the interactive shell displays them using forward slashes, as open source software developers have historically favored the Linux operating system.

If you want to get a simple text string of this path, you can pass it to the `str()` function, which in our example returns `'spam\\bacon\\eggs'`. (Notice that we double the backslashes because we need to escape each backslash with another backslash character.) If I had called this function on macOS or Linux, `Path()` would have returned a `PosixPath` object that, when passed to `str()`, would have returned `'spam/bacon/eggs'`. (*POSIX* is a set of standards for Unix-like operating systems.)

If you work with `Path` objects, `WindowsPath` and `PosixPath` never have to appear in your source code directly. These `Path` objects will be passed to several of the file-related functions introduced in this chapter. For example, the following code joins names from a list of filenames to the end of a folder's name:

```
>>> from pathlib import Path
>>> my_files = ['accounts.txt', 'details.csv', 'invite.docx']
>>> for filename in my_files:
...     print(Path(r'C:\Users\Al', filename))
...
C:\Users\Al\accounts.txt
C:\Users\Al\details.csv
C:\Users\Al\invite.docx
```

On Windows, the backslash separates directories, so you can't use it in filenames. However, you can use backslashes in filenames on macOS and Linux. So, while `Path(r'spam\eggs')` refers to two separate folders (or a file *eggs* in a folder *spam*) on Windows, the same command would refer to a single folder (or file) named *spam\eggs* on macOS and Linux. For this reason, it's usually a good idea to always use forward slashes in your Python code (and I'll be doing so for the rest of this chapter). The `pathlib` module will ensure that your code always works on all operating systems.

Joining Paths

We normally use the `+` operator to add two integer or floating-point numbers, such as in the expression `2 + 2`, which evaluates to the integer value `4`. But we can also use the `+` operator to concatenate two string values, like the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

expression `'Hello' + 'World'`, which evaluates to the string value `'HelloWorld'`. Similarly, the / operator that we normally use for division can combine `Path` objects and strings. This is helpful for modifying a `Path` object after you've already created it with the `Path()` function.

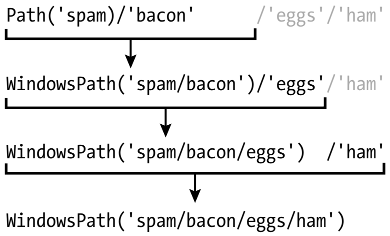
For example, enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('spam') / 'bacon' / 'eggs'
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon/eggs')
WindowsPath('spam/bacon/eggs')
>>> Path('spam') / Path('bacon', 'eggs')
WindowsPath('spam/bacon/eggs')
```

The only thing you need to keep in mind when using the / operator for joining paths is that one of the first two values in the expression must be a `Path` object. This is because these expressions evaluate from left to right, and the / operator can be used on two `Path` objects or on a `Path` object and a string, but not on two strings. Python will give you an error if you try to enter the following into the interactive shell:

```
>>> 'spam' / 'bacon'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

So, either the first or second leftmost value must be a `Path` object for the entire expression to evaluate to a `Path` object. Here's how the / operator and a `Path` object evaluate to the final `Path` object:


Path('spam)/'bacon'   /'eggs'/'ham'
WindowsPath('spam/bacon')/'eggs'/'ham'
WindowsPath('spam/bacon/eggs') /'ham'
WindowsPath('spam/bacon/eggs/ham')

If you see the `TypeError: unsupported operand type(s) for /: 'str' and 'str'` error message shown previously, you need to put a `Path` object instead of a string on the left side of the expression.

The / operator replaces the older `os.path.join()` function, which you can learn more about at *https://docs.python.org/3/library/os.path.html#os.path.join.*

Accessing the Current Working Directory

Every program that runs on your computer has a *current working directory*. Any filenames or paths that do not begin with the root folder are assumed to be under the current working directory.

NOTE *While* folder *is the more modern name for* directory, *note that* current working directory *(or just* working directory*) is the standard term, not current working* folder.

You can get the current working directory as a string value with the `Path.cwd()` function and can change it using `os.chdir()`. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> import os
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python313')'
>>> os.chdir('C:\\Windows\\System32')
>>> Path.cwd()
WindowsPath('C:/Windows/System32')
```

Here, the current working directory is set to *C:\Users\Al\AppData\Local\Programs\Python\Python313*, so the filename *project.docx* refers to *C:\Users\Al\AppData\Local\Programs\Python\Python313\project.docx*. When we change the current working directory to *C:\Windows\System32*, the filename *project.docx* is interpreted as *C:\Windows\System32\project.docx*.

Python will display an error if you try to change to a directory that does not exist:

```
>>> import os
>>> os.chdir('C:/ThisFolderDoesNotExist')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
FileNotFoundError: [WinError 2] The system cannot find the file specified:
'C:/ThisFolderDoesNotExist'
```

There is no `pathlib` function for changing the working directory. You must use `os.chdir()`.

The `os.getcwd()` function is the older way of getting the current working directory as a string. It's documented at *https://docs.python.org/3/library/os.html#os.getcwd*.

Accessing the Home Directory

All users have a folder for their own files on their computer; this folder is called the *home directory* or *home folder*. You can get a `Path` object of the home folder by calling `Path.home()`:

```
>>> from pathlib import Path
>>> Path.home()
WindowsPath('C:/Users/Al')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The home directories are located in a set place depending on your operating system:

- On Windows, home directories are under *C:\Users*.
- On macOS, home directories are under */Users*.
- On Linux, home directories are often under */home*.

Your scripts will almost certainly have permissions to read and write the files under your home directory, so it's an ideal place to put the files that your Python programs will work with.

Specifying Absolute vs. Relative Paths

There are two ways to specify a filepath:

- An *absolute path*, which always begins with the root folder (*C:\* on Windows and */* on macOS and Linux)
- A *relative path*, which is relative to the program's current working directory

On Windows, *C:\* is the root for the main hard drive. This lettering dates back to the 1960s, when computers had two floppy disk drives labeled *A:\* and *B:\*. On Windows, USB flash memory and DVD drives are assigned to letters *D:\* and higher. Use one of these drives as the root folder to access files on that storage media.

There are also the *dot* (.) and *dot-dot* (..) folders. These are not real folders but special names that can be used in a filepath. A single period (dot) for a folder name is shorthand for *this folder*. Two periods (dot-dot) means *the parent folder*.

Figure 10-2 shows some example folders and files. When the current working directory is set to *C:\bacon*, the relative paths for the other folders and files are set as they are in the figure.

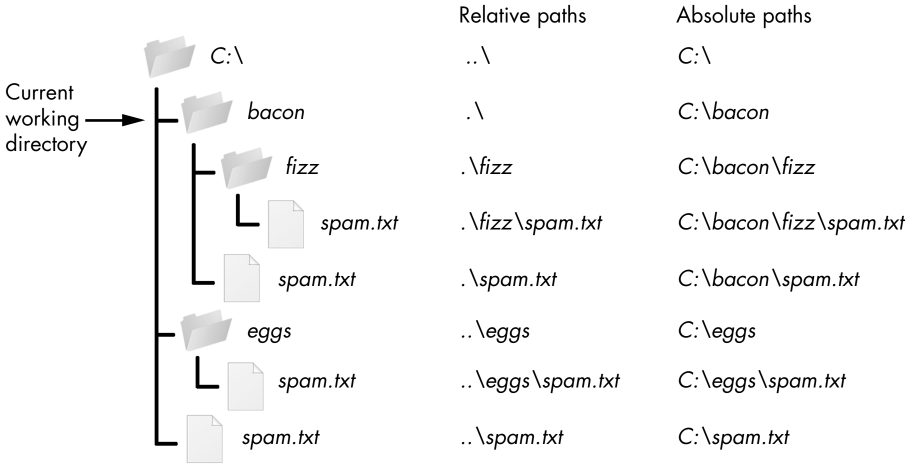


Figure 10-2: The relative paths for folders and files in the working directory C:\bacon

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The .\ at the start of a relative path is optional. For example, *.\spam.txt* and *spam.txt* refer to the same file.

Creating New Folders

Your programs can create new folders with the `os.makedirs()` function. Enter the following into the interactive shell:

```
>>> import os
>>> os.makedirs('C:\\delicious\\walnut\\waffles')
```

This will create not just the *C:\delicious* folder but also a *walnut* folder inside *C:\delicious* and a *waffles* folder inside *C:\delicious\walnut*. That is, `os.makedirs()` will create any necessary intermediate folders to ensure that the full path exists. Figure 10-3 shows this hierarchy of folders.

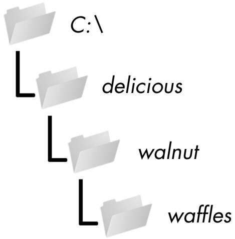


Figure 10-3: The result of os.makedirs('C:\\delicious\\walnut\\waffles')

To make a directory from a `Path` object, call the `mkdir()` method. For example, this code will create a *spam* folder under the home folder on my computer:

```
>>> from pathlib import Path
>>> Path(r'C:\Users\Al\spam').mkdir()
```

Note that `mkdir()` can only make one directory at a time; it won't make several subdirectories at once, like `os.makedirs()` will.

Handling Absolute and Relative Paths

Calling the `is_absolute()` method on a `Path` object will return `True` if it represents an absolute path or `False` if it represents a relative path. For example, enter the following into the interactive shell, using your own files and folders instead of the exact ones listed here:

```
>>> from pathlib import Path
>>> Path.cwd()
WindowsPath('C:/Users/Al/AppData/Local/Programs/Python/Python312')
>>> Path.cwd().is_absolute()
True
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> Path('spam/bacon/eggs').is_absolute()
False
```

To get an absolute path from a relative path, you can put `Path.cwd() /` in front of the relative `Path` object. After all, when we say "relative path," we almost always mean a path that is relative to the current working directory. The `absolute()` method also returns this `Path` object. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.cwd() / Path('my/relative/path')
WindowsPath('C:/Users/Al/Desktop/my/relative/path')
>>> Path('my/relative/path').absolute()
WindowsPath('C:/Users/Al/Desktop/my/relative/path')
```

If your relative path is relative to another path besides the current working directory, replace `Path.cwd()` with that other path. The following example gets an absolute path using the home directory instead of the current working directory:

```
>>> from pathlib import Path
>>> Path('my/relative/path')
WindowsPath('my/relative/path')
>>> Path.home() / Path('my/relative/path')
WindowsPath('C:/Users/Al/my/relative/path')
```

`Path` objects are used to represent both relative and absolute paths. The only difference is whether the `Path` object begins with the root folder or not.

Getting the Parts of a Filepath

Given a `Path` object, you can extract the filepath's different parts as strings using several `Path` object attributes. These can be useful for constructing new filepaths based on existing ones. Figure 10-4 diagrams the attributes.

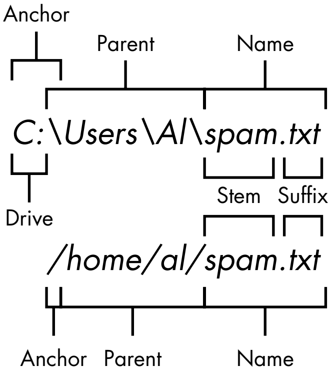


Figure 10-4: The parts of a Windows (top) and macOS/Linux (bottom) filepath

The parts of a filepath include the following:

- The *anchor*, which is the root folder of the filesystem
- On Windows, the *drive*, which is the single letter that often denotes a physical hard drive or other storage device
- The *parent*, which is the folder that contains the file
- The *name* of the file, made up of the *stem* (or *base name*) and the *suffix* (or *extension*)

Note that Windows `Path` objects have a `drive` attribute, but macOS and Linux `Path` objects don't. The `drive` attribute doesn't include the first backslash.

To extract each attribute from the filepath, enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.anchor
'C:\\'
>>> p.parent
WindowsPath('C:/Users/Al')
>>> p.name
'spam.txt'
>>> p.stem
'spam'
>>> p.suffix
'.txt'
>>> p.drive
'C:'
```

These attributes evaluate to simple string values, except for parent, which evaluates to another Path object. If you want to split up a path by its separator, access the parts attribute to get a tuple of string values:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/spam.txt')
>>> p.parts
('C:\\', 'Users', 'Al', 'spam.txt')
 >>> p.parts[3]
'spam.txt'
>>> p.parts[0:2]
('C:\\', 'Users')
```

Note that even though the string used in the Path() call contains forward slashes, parts uses an anchor on Windows that has the appropriate backslash: 'C:\\' (or r'C:\' as a raw string with the backslash unescaped).

The parents attribute (which is different from the parent attribute) evaluates to the ancestor folders of a Path object with an integer index:

```
>>> from pathlib import Path
>>> Path.cwd()
WindowsPath('C:/Users/Al/Desktop')
>>> Path.cwd().parents[0]
WindowsPath('C:/Users/Al')
>>> Path.cwd().parents[1]
WindowsPath('C:/Users')
>>> Path.cwd().parents[2]
WindowsPath('C:/')
```

If you keep following the parent folders, you will end up with the root folder.

Finding File Sizes and Timestamps

Once you have ways to handle filepaths, you can start gathering information about specific files and folders. The stat() method returns a stat_result object with file size and timestamp information about a file.

For example, enter the following into the interactive shell to find out about the *calc.exe* program file on Windows:

```
>>> from pathlib import Path
>>> calc_file = Path('C:/Windows/System32/calc.exe')
>>> calc_file.stat()
os.stat_result(st_mode=33279, st_ino=562949956525418, st_dev=3739257218,
st_nlink=2, st_uid=0, st_gid=0, st_size=27648, st_atime=1678984560,
st_mtime=1575709787, st_ctime=1575709787)
>>> calc_file.stat().st_size
27648
>>> calc_file.stat().st_mtime
1712627129.0906117
>>> import time
>>> time.asctime(time.localtime(calc_file.stat().st_mtime))
'Mon Apr  8 20:45:29 2024'
```

The `st_size` attribute of the `stat_result` object returned by the `stat()` method is the size of the file in bytes. You can divide this integer by `1024`, by `1024 ** 2`, or by `1024 ** 3` to get the size in KB, MB, or GB, respectively.

The `st_mtime` is the "last modified" timestamp, which can be useful for, say, figuring out the last time a *.docx* Word file was changed. This timestamp is in Unix epoch time, which is the number of seconds since January 1, 1970. The `time` module (explained in Chapter 19) has functions for turning this number into a human-readable form.

The `stat_result` object has several useful attributes:

`st_size`

The size of the file in bytes.

`st_mtime`

The "last modified" timestamp, when the file was last changed.

`st_ctime`

The "creation" timestamp. On Windows, this identifies when the file was created. On macOS and Linux, this identifies the last time the file's metadata (such as its name) was changed.

`st_atime`

The "last accessed" timestamp, when the file was last read.

Keep in mind that the modified, creation, and access timestamps can be changed manually, and are not guaranteed to be accurate.

Finding Files Using Glob Patterns

The `*` and `?` characters can be used to match folder names and filenames in what are called *glob patterns*. Glob patterns are like a simplified regex language: the `*` character matches any text, and the `?` character matches exactly one character. For example, look at these glob patterns:

`'*.txt'` matches all files that end with *.txt*.

`'project?.txt'` matches `'project1.txt'`, `'project2.txt'`, or `'projectX.txt'`.

`'*project?.*'` matches `'catproject5.txt'` or `'secret_project7.docx'`.

`'*'` matches all filenames.

`Path` objects of folders have a `glob()` method for listing any content in the folder that matches the glob pattern. The `glob()` method returns a generator object (the topic of which is beyond the scope of this book) that you'll need to pass to `list()` to easily view in the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('C:/Users/Al/Desktop')
>>> p.glob('*')
<generator object Path.glob at 0x000002A6E389DED0>
>>> list(p.glob('*'))
```

```
[WindowsPath('C:/Users/Al/Desktop/1.png'), WindowsPath('C:/Users/Al/
Desktop/22-ap.pdf'), WindowsPath('C:/Users/Al/Desktop/cat.jpg'),
WindowsPath('C:/Users/Al/Desktop/zzz.txt')]
```

You can also use the generator object that glob() returns in a for loop:

```
>>> from pathlib import Path
>>> for name in Path('C:/Users/Al/Desktop').glob('*'):
>>>     print(name)
C:\Users\Al\Desktop\1.png
C:\Users\Al\Desktop\22-ap.pdf
C:\Users\Al\Desktop\cat.jpg
C:\Users\Al\Desktop\zzz.txt
```

If you want to perform an operation on every file in a folder, such as copying it to a backup folder or renaming it, the glob('*') method call can get you the list of Path objects for these files and folders. Note that glob patterns are also commonly used in command line commands such as ls or dir. Chapter 12 discusses the command line in more detail.

Checking Path Validity

Many Python functions will crash with an error if you supply them with a path that does not exist. Luckily, Path objects have methods to check whether a given path exists and whether it is a file or folder. Assuming that a variable p holds a Path object, you could expect the following:

- Calling p.exists() returns True if the path exists, and returns False if it doesn't exist.
- Calling p.is_file() returns True if the path exists and is a file, and returns False otherwise.
- Calling p.is_dir() returns True if the path exists and is a directory, and returns False otherwise.

On my computer, here is what I get when I try these methods in the interactive shell:

```
>>> from pathlib import Path
>>> win_dir = Path('C:/Windows')
>>> not_exists_dir = Path('C:/This/Folder/Does/Not/Exist')
>>> calc_file_path = Path('C:/Windows
/System32/calc.exe')
>>> win_dir.exists()
True
>>> win_dir.is_dir()
True
>>> not_exists_dir.exists()
False
>>> calc_file_path.is_file()
True
>>> calc_file_path.is_dir()
False
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You can determine whether there is a DVD or flash drive currently attached to the computer by checking for it with the `exists()` method. For instance, if I wanted to check for a flash drive with the volume named *D:\* on my Windows computer, I could do that with the following:

```
>>> from pathlib import Path
>>> d_drive = Path('D:/')
>>> d_drive.exists()
False
```

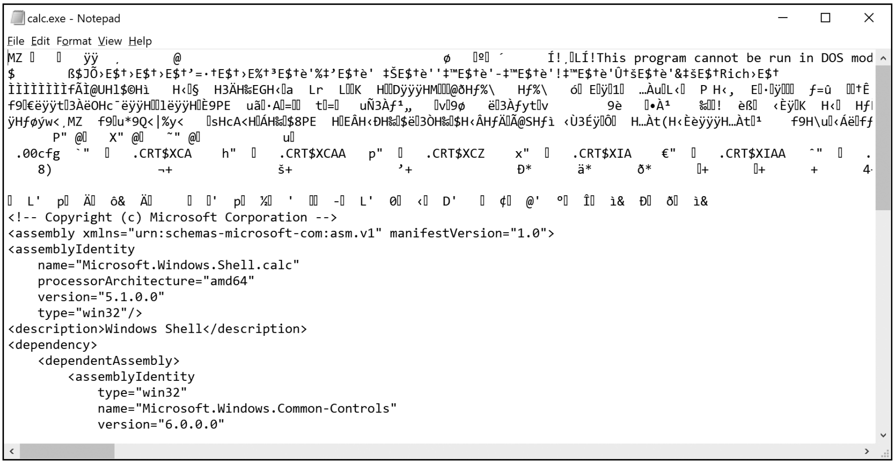
Oops! It looks like I forgot to plug in my flash drive.

The older `os.path` module can accomplish the same task with the `os.path.exists(`*path*`)`, `os.path.isfile(`*path*`)`, and `os.path.isdir(`*path*`)` functions, which act just like their `Path` function counterparts. As of Python 3.6, these functions can accept `Path` objects as well as strings of the filepaths.

The File Reading and Writing Process

Once you're comfortable working with folders and relative paths, you'll be able to specify the locations of files to read and write. The functions covered in the next few sections apply to plaintext files. *Plaintext files* contain only basic text characters and do not include font, size, or color information. Text files with the *.txt* extension or Python script files with the *.py* extension are examples of plaintext files. You can open these with the Windows Notepad or macOS TextEdit application, and your programs can easily read their content, then treat it as an ordinary string value.

Binary files are all other file types, such as word processing documents, PDFs, images, spreadsheets, and executable programs. If you open a binary file in Notepad or TextEdit, it will look like scrambled nonsense, like that shown in Figure 10-5.



```
MZ □    □   ÿÿ  ¸        @                                        ø    □º□ ´      Í!¸□LÍ!This program cannot be run in DOS mod
$       ß$JÕ›E$†›E$†›E$†’=·†E$†›E%†³E$†è'%‡’E$†è' ‡ŠE$†è''‡™E$†è'-‡™E$†è'!‡™E$†è'Û†šE$†è'&‡šE$†Rich›E$†
ÌÌÌÌÌÌÌÌfÃÌ@UHl$©Hì   H‹□§  H3ÄH‰EGH‹□a  Lr  L□□K  H□□DÿÿÿHM□□□@ðHƒ%\    Hƒ%\    ó□ E□ÿ□1□  …Àu□L‹□  P H‹,  E□·□ÿ□□□  ƒ=û  □□†Ê
f9□€ëÿÿt□3ÀëOHc¯ëÿÿH□□lëÿÿH□È9PE  uã□·A□=□□  t□=□   uÑ3Àƒ¹„     □v□9ø    ë□3Àƒyt□v         9è    □•À¹     ‰□□!  èß□  ‹Èÿ□K  H‹□  Hƒ
ÿHƒøýw‹¸MZ  f9□u*9Q<|%y<   □sHcA<H□ÁH‰□$8PE  H□EÂH‹ÐH‰□$ë□3ÒH‰□$H‹ÂHƒÄ□Ã@SHƒì ‹Ù3Éÿ□Ô□  H…Àt(H‹ÈèÿÿÿH…Àt□¹     f9H\u□‹Áë□ƒƒ
      P" @□    X" @□    ˜" @□            u□
 .00cfg  `"   □   .CRT$XCA     h"   □   .CRT$XCAA    p"   □   .CRT$XCZ     x"   □   .CRT$XIA     €"   □   .CRT$XIAA    ˆ"   □   .
    8)              ¬+                 š+                ’+                Ð*        ä*        ð*        □+        □+        +        4

□  L'   p□   Ä□   ô&   Ä□      □   □'   p□   ½□   '   □□   -□   L'   0□   ‹□   D'    □   ¢□   @'   °□   Î□   ì&   Ð□   ð□   ì&
<!-- Copyright (c) Microsoft Corporation -->
<assembly xmlns="urn:schemas-microsoft-com:asm.v1" manifestVersion="1.0">
<assemblyIdentity
    name="Microsoft.Windows.Shell.calc"
    processorArchitecture="amd64"
    version="5.1.0.0"
    type="win32"/>
<description>Windows Shell</description>
<dependency>
    <dependentAssembly>
        <assemblyIdentity
            type="win32"
            name="Microsoft.Windows.Common-Controls"
            version="6.0.0.0"
```

Figure 10-5: The Windows calc.exe *program opened in Notepad*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Because we must handle each type of binary file in its own way, this book won't discuss reading and writing raw binary files directly. Fortunately, many modules make working with binary files easier, and you'll explore one of them, the `shelve` module, later in this chapter.

The `pathlib` module's `read_text()` method returns the full contents of a text file as a string. Its `write_text()` method creates a new text file (or overwrites an existing one) with the string passed to it. Enter the following into the interactive shell:

```
>>> from pathlib import Path
>>> p = Path('spam.txt')
>>> p.write_text('Hello, world!')
13
>>> p.read_text()
'Hello, world!'
```

These method calls create a *spam.txt* file with the content `'Hello, world!'`. The `13` that `write_text()` returns indicates that 13 characters were written to the file. (You can often disregard this return value.) The `read_text()` call reads and returns the contents of the new file as a string: `'Hello, world!'`.

Keep in mind that these `Path` object methods allow only basic interactions with files. The more common way of writing to a file involves using the `open()` function and file objects. There are three steps to reading or writing files in Python:

1. Call the `open()` function to return a `File` object.
2. Call the `read()` or `write()` method on the `File` object.
3. Close the file by calling the `close()` method on the `File` object.

We'll go over these steps in the following sections.

Note that as you begin working with files, you may find it helpful to be able to quickly see their extensions (*.txt*, *.pdf*, *.jpg*, and so on). Windows and macOS may hide file extensions by default, showing *spam.txt* as simply *spam*. To show extensions, open the settings for File Explorer (on Windows) or Finder (on macOS) and look for a checkbox that says something like "Show all filename extensions" or "Hide extensions for known file types." (The exact location and wording of this setting depend on the version of your operating system.)

Opening Files

To open a file with the `open()` function, pass it a string path indicating the file you want to open. This can be either an absolute path or a relative path. The `open()` function returns a `File` object.

Try this by creating a text file named *hello.txt* using Notepad or TextEdit. Enter **`Hello, world!`** as the content of this text file and save it in your user home folder. Then, enter the following in the interactive shell:

```
>>> from pathlib import Path
>>> hello_file = open(Path.home() / 'hello.txt', encoding='UTF-8')
```

The open() function will open the file in "reading plaintext" mode, or *read mode* for short. When a file is opened in read mode, Python lets you read the file's data but not write or modify it in any way. Read mode is the default mode for files you open in Python. But if you don't want to rely on Python's defaults, you can explicitly specify the mode by passing the string value 'r' as a second argument to open(). For example, open('/Users/Al/hello.txt', 'r') does the same thing as open('/Users/Al/hello.txt').

The encoding named parameter specifies what encoding to use when converting the bytes in the file to a Python text string. The correct encoding is almost always 'utf-8', which is also the default encoding used on macOS and Linux. However, Windows uses 'cp1252' for its default encoding (also known as *extended ASCII*). This can cause problems when trying to read certain UTF-8 encoded text files with non-English characters on Windows, so it's a good habit to pass encoding='utf-8' to your open() function calls when opening files in plaintext read, write, or append mode. The binary read, write, and append modes don't use the encoding named parameter, so you can leave it out in those cases.

The call to open() returns a File object. A File object represents a file on your computer; it is simply another type of value in Python, much like the lists and dictionaries you're already familiar with. In the previous example, you stored the File object in the variable hello_file. Now, whenever you want to read from or write to the file, you can do so by calling methods on the File object in hello_file.

Reading the Contents of Files

Now that you have a File object, you can start reading from it. If you want to read the entire contents of a file as a string value, use the File object's read() method. Let's continue with the *hello.txt* File object you stored in hello_file. Enter the following into the interactive shell:

```
>>> hello_content = hello_file.read()
>>> hello_content
'Hello, world!'
```

You can think of the contents of a file as a single large string value; the read() method merely returns the string that is stored in the file.

Alternatively, you can use the readlines() method to get a *list* of string values from the file, one for each line of text. For example, create a file named *sonnet29.txt* in the same directory as *hello.txt* and place the following text in it:

```
When, in disgrace with fortune and men's eyes,
I all alone beweep my outcast state,
And trouble deaf heaven with my bootless cries,
And look upon myself and curse my fate,
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Make sure to separate the four lines with line breaks. Then, enter the following into the interactive shell:

```
>>> sonnet_file = open(Path.home() / 'sonnet29.txt', encoding='UTF-8')
>>> sonnet_file.readlines()
["When, in disgrace with fortune and men's eyes,\n", 'I all alone beweep
my outcast state,\n', And trouble deaf heaven with my bootless cries,\n',
'And look upon myself and curse my fate,']
```

Note that, except for the last line of the file, each of the string values ends with a newline character \n. A list of strings is often easier to work with than a single large string value.

Writing to Files

Python allows you to write content to a file, just as the print() function writes strings to the screen. You can't write to a file you've opened in read mode, though. Instead, you need to open it in "write plaintext" mode or "append plaintext" mode, called *write mode* and *append mode* for short.

Write mode will overwrite the existing file, which is similar to overwriting a variable's value with a new value. Pass 'w' as the second argument to open() to open the file in write mode. Append mode, on the other hand, will append text to the end of the existing file. You can think of this mode as appending values to a list in a variable rather than overwriting the variable altogether. Pass 'a' as the second argument to open() to open the file in append mode.

If the filename passed to open() does not exist, both write mode and append mode will create a new, blank file. After reading or writing a file, call the close() method before opening the file again.

Let's put these concepts together. Enter the following into the interactive shell:

```
>>> bacon_file = open('bacon.txt', 'w', encoding='UTF-8')
>>> bacon_file.write('Hello, world!\n')
14
>>> bacon_file.close()
>>> bacon_file = open('bacon.txt', 'a', encoding='UTF-8')
>>> bacon_file.write('Bacon is not a vegetable.')
25
>>> bacon_file.close()
>>> bacon_file = open('bacon.txt', encoding='UTF-8')
>>> content = bacon_file.read()
>>> bacon_file.close()
>>> print(content)
Hello, world!
Bacon is not a vegetable.
```

First, we open *bacon.txt* in write mode. As no *bacon.txt* file exists yet, Python creates one. Calling write() on the opened file and passing write() the string argument 'Hello, world!\n' writes the string to the file and

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

returns the number of characters written, including the newline. Then, we close the file.

To add text to the existing contents of the file instead of replacing the string we just wrote, we open the file in append mode. We write `'Bacon is not a vegetable.'` to the file and close it. Finally, to print the file contents to the screen, we open the file in its default read mode, call `read()`, store the resulting `File` object in `content`, close the file, and print `content`.

Note that the `write()` method does not automatically add a newline character to the end of the string like the `print()` function does. You will have to add this character yourself.

You can also pass a `Path` object to the `open()` function instead of the filename as a string.

Using with Statements

Every file on which your program calls `open()` needs `close()` called on it as well, but you may forget to include the `close()` function, or your program might skip over the `close()` call in certain circumstances.

Python's `with` statement makes it easier to automatically close files. A `with` statement creates something called a *context manager* that Python uses to manage resources. These resources, such as files, network connections, or segments of memory, often have setup and teardown steps during which the resource is allocated and later released so that other programs can make use of it. (Most of the time, however, you'll encounter `with` statements used to open files.)

The `with` statement adds a block of code that begins by allocating the resource and then releases it when the program execution leaves the block, which could happen due to a `return` statement, an unhandled exception being raised, or some other reason.

Here is typical code that writes and reads the content of a file:

```
file_obj = open('data.txt', 'w', encoding='utf-8')
file_obj.write('Hello, world!')
file_obj.close()
file_obj = open('data.txt', encoding='utf-8')
content = file_obj.read()
file_obj.close()
```

Here is the equivalent code using a `with` statement:

```
with open('data.txt', 'w', encoding='UTF-8') as file_obj:
    file_obj.write('Hello, world!')
with open('data.txt', encoding='UTF-8') as file_obj:
    content = file_obj.read()
```

In the `with` statement example, notice that there are no calls to `close()` at all because the `with` statement automatically calls it when the program execution leaves the block. The `with` statement knows to do this based on the context manager it obtains from the `open()` function. Creating your own

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

context managers is beyond the scope of this book, but you can learn about them from the online documentation at *https://docs.python.org/3/reference/datamodel.html#context-managers* or the book *Serious Python* by Julien Danjou (No Starch Press, 2018).

Saving Variables with the shelve Module

You can save variables in your Python programs to binary shelf files using the `shelve` module. This lets your program restore that data to the variables the next time it is run. You could use this technique to add Save and Open features to your program; for example, if you ran a program and entered some configuration settings, you could save those settings to a shelf file and then have the program load the settings the next time it is run.

To practice using `shelve`, enter the following into the interactive shell:

```
>>> import shelve
>>> shelf_file = shelve.open('mydata')

>>> shelf_file['cats'] = ['Zophie', 'Pooka', 'Simon']
>>> shelf_file.close()
```

To read and write data using the `shelve` module, you first import `shelve`. Next, call `shelve.open()` and pass it a filename, then store the returned shelf value in a variable. You can make changes to the shelf value as if it were a dictionary. When you're done, call `close()` on the shelf value. Here, our shelf value is stored in `shelf_file`. We create a list `cats` and write `shelf_file['cats'] = ['Zophie', 'Pooka', 'Simon']` to store the list in `shelf_file` as a value associated with the key `'cats'` (like in a dictionary). Then, we call `close()` on `shelf_file`.

After running the previous code on Windows, you should see three new files in the current working directory: *mydata.bak*, *mydata.dat*, and *mydata.dir*. On macOS, you should see only a single *mydata.db* file, and Linux has a single *mydata* file. These binary files contain the data you stored in your shelf. The format of these binary files isn't important; you only need to know what the `shelve` module does, not how it does it. The module frees you from worrying about how to store your program's data to a file.

Your programs can use the `shelve` module to later reopen and retrieve the data from these shelf files. Shelf values don't have to be opened in read or write mode; they allow both reading and writing once opened. Enter the following into the interactive shell:

```
>>> shelf_file = shelve.open('mydata')
>>> type(shelf_file)
<class 'shelve.DbfilenameShelf'>
>>> shelf_file['cats']
['Zophie', 'Pooka', 'Simon']
>>> shelf_file.close()
```

Here, we open the shelf files to check that they stored the data correctly. Entering shelf_file['cats'] returns the same list we created earlier. Now that we know the file stored the list correctly, we call close().

Just like dictionaries, shelf values have keys() and values() methods that will return list-like values of the keys and values in the shelf. Because these return values are not true lists, you should pass them to the list() function to get them in list form. Enter the following into the interactive shell:

```
>>> shelf_file = shelve.open('mydata')
>>> list(shelf_file.keys())
['cats']
>>> list(shelf_file.values())
[['Zophie', 'Pooka', 'Simon']]
>>> shelf_file.close()
```

Plaintext is useful for creating files that you'll read in a text editor such as Notepad or TextEdit, but if you want to save data from your Python programs, use the shelve module.

Project 4: Generate Random Quiz Files

Say you're a geography teacher with 35 students in your class and you want to give a pop quiz on US state capitals. Alas, your class has a few bad eggs in it, and you can't trust the students not to cheat. You'd like to randomize the order of questions so that each quiz is unique, making it impossible for anyone to crib answers from anyone else. Of course, doing this by hand would be a lengthy and boring affair. Fortunately, you know some Python.

Here is what the program does:

- Creates 35 different quizzes
- Creates 50 multiple-choice questions for each quiz, in random order
- Provides the correct answer and three random wrong answers for each question, in random order
- Writes the quizzes to 35 text files
- Writes the answer keys to 35 text files

This means the code will need to do the following:

- Store the states and their capitals in a dictionary.
- Call open(), write(), and close() for the quiz and answer key text files.
- Use random.shuffle() to randomize the order of the questions and multiple-choice options.

Let's get started.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Step 1: Store the Quiz Data in a Dictionary

The first step is to create a skeleton script and fill it with your quiz data. Create a file named *randomQuizGenerator.py*, and make it look like the following:

```
  # randomQuizGenerator.py - Creates quizzes with questions and answers in
  # random order, along with the answer key

❶ import random

  # The quiz data. Keys are states and values are their capitals.
❷ capitals = {'Alabama': 'Montgomery', 'Alaska': 'Juneau', 'Arizona':
  'Phoenix', 'Arkansas': 'Little Rock', 'California': 'Sacramento', 'Colorado':
  'Denver', 'Connecticut': 'Hartford', 'Delaware': 'Dover', 'Florida':
  'Tallahassee', 'Georgia': 'Atlanta', 'Hawaii': 'Honolulu', 'Idaho': 'Boise',
  'Illinois': 'Springfield', 'Indiana': 'Indianapolis', 'Iowa': 'Des Moines',
  'Kansas': 'Topeka', 'Kentucky': 'Frankfort', 'Louisiana': 'Baton Rouge',
  'Maine': 'Augusta', 'Maryland': 'Annapolis', 'Massachusetts': 'Boston',
  'Michigan': 'Lansing', 'Minnesota': 'Saint Paul', 'Mississippi': 'Jackson',
  'Missouri': 'Jefferson City', 'Montana': 'Helena', 'Nebraska': 'Lincoln',
  'Nevada': 'Carson City', 'New Hampshire': 'Concord', 'New Jersey': 'Trenton',
  'New Mexico': 'Santa Fe', 'New York': 'Albany', 'North Carolina': 'Raleigh',
  'North Dakota': 'Bismarck', 'Ohio': 'Columbus', 'Oklahoma': 'Oklahoma City',
  'Oregon': 'Salem', 'Pennsylvania': 'Harrisburg', 'Rhode Island': 'Providence',
  'South Carolina': 'Columbia', 'South Dakota': 'Pierre', 'Tennessee':
  'Nashville', 'Texas': 'Austin', 'Utah': 'Salt Lake City', 'Vermont':
  'Montpelier', 'Virginia': 'Richmond', 'Washington': 'Olympia', 'West
  Virginia':'Charleston', 'Wisconsin': 'Madison', 'Wyoming': 'Cheyenne'}

  # Generate 35 quiz files.
❸ for quiz_num in range(35):
      # TODO: Create the quiz and answer key files.

      # TODO: Write out the header for the quiz.

      # TODO: Shuffle the order of the states.

      # TODO: Loop through all 50 states, making a question for each.
```

Because this program will randomly order the questions and answers, you'll need to import the `random` module ❶ to make use of its functions. The `capitals` variable ❷ contains a dictionary with US states as keys and their capitals as values. And because you want to create 35 quizzes, the code that actually generates the quiz and answer key files (marked with `TODO` comments for now) will go inside a `for` loop that loops 35 times ❸. (You can change this number to generate any number of quiz files.)

Step 2: Create the Quiz File

Now it's time to start filling in those `TODO`s.

The code in the loop will repeat 35 times, once for each quiz, so you have to worry about only one quiz at a time within the loop. First, you'll create the actual quiz file. It needs a unique filename and some kind of standard header, with places for the student to fill in a name, date, and class period. Then, you'll need to get a list of states in randomized order, which you can use later to create the questions and answers for the quiz.

Add the following lines of code to *randomQuizGenerator.py*:

```
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key

--snip--

# Generate 35 quiz files.
for quiz_num in range(35):
    # Create the quiz and answer key files.
    quiz_file = open(f'capitalsquiz{quiz_num + 1}.txt', 'w', encoding='UTF-8') ❶
    answer_file = open(f'capitalsquiz_answers{quiz_num + 1}.txt', 'w', encoding='UTF-8') ❷

    # Write out the header for the quiz.
    quiz_file.write('Name:\n\nDate:\n\nPeriod:\n\n') ❸
    quiz_file.write((' ' * 20) + f'State Capitals Quiz (Form{quiz_num + 1})')
    quiz_file.write('\n\n')

    # Shuffle the order of the states.
    states = list(capitals.keys())
    random.shuffle(states) ❹

    # TODO: Loop through all 50 states, making a question for each.
```

The quizzes will use the filenames *capitalsquiz<N>.txt*, where *<N>* is a unique number that comes from `quiz_num`, the `for` loop's counter. We'll store the answer key for *capitalsquiz<N>.txt* in the text files named *capitalsquiz_answers<N>.txt*. On each iteration of the loop, the code will replace the `{quiz_num + 1}` placeholders in these filenames with a unique number. For example, it names the first quiz and answer key *capitalsquiz1.txt* and *capitalsquiz_answers1.txt*. We create these files with calls to the `open()` function at ❶ and ❷, passing `'w'` as the second argument to open them in write mode.

The `write()` statements at ❸ create a quiz header for the student to fill out. Finally, we generate a randomized list of US states with the help of the `random.shuffle()` function ❹, which randomly reorders the values in any list that is passed to it.

Step 3: Create the Answer Options

Now you need to generate answer options A to D for each question using another `for` loop. Later, a third, nested `for` loop will write these multiple-choice options to the files. Make your code look like the following:

```
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key

--snip--
```

```
    # Loop through all 50 states, making a question for each.
    for num in range(50):

        # Get right and wrong answers.
        correct_answer = capitals[states[num]]
        wrong_answers = list(capitals.values())
        del wrong_answers[wrong_answers.index(correct_answer)]
        wrong_answers = random.sample(wrong_answers, 3)
        answer_options = wrong_answers + [correct_answer]
        random.shuffle(answer_options)

        # TODO: Write the question and answer options to the quiz file.

        # TODO: Write the answer key to a file.
```

The correct answer is easy to create; it's already stored as a value in the capitals dictionary. This loop will iterate through the states in the shuffled states list, find each state in capitals, and store that state's corresponding capital in correct_answer.

Creating the list of possible wrong answers is trickier. You can get it by duplicating the values in the capitals dictionary, deleting the correct answer, and selecting three random values from this list. The random.sample() function makes performing this selection easy. Its first argument is the list you want to select from, and the second argument is the number of values to select. The full list of answer options combines these three wrong answers with the correct answers. Finally, we randomize the answers so that the correct response isn't always choice D.

Step 4: Write the Content to the Files

All that is left is to write the question to the quiz file and the answer to the answer key file. Make your code look like the following:

```
# randomQuizGenerator.py - Creates quizzes with questions and answers in
# random order, along with the answer key

--snip--

    # Loop through all 50 states, making a question for each.
    for num in range(50):
        --snip--

        # Write the question and the answer options to the quiz file.
        quiz_file.write(f'{num + 1}. Capital of {states[num]}:\n')
        for i in range(4): ❶
            quiz_file.write(f"    {❷ 'ABCD'[i]}. { answer_options[i]}\n")
        quiz_file.write('\n')

        # Write the answer key to a file.
        answer_file.write(f"{num + 1}.{'ABCD'[answer_options.index(correct_answer)]}") ❸
    quiz_file.close()
    answer_file.close()
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

A `for` loop iterates through integers `0` to `3` to write the answer options in the `answer_options` list to the file ❶. The expression `'ABCD'[i]` at ❷ treats the string `'ABCD'` as an array and will evaluate to `'A'`,`'B'`, `'C'`, and `'D'` on each respective iteration through the loop.

In the final line of the loop, the expression `answer_options.index(correct_answer)` ❸ will find the integer index of the correct answer in the randomly ordered answer options, causing the correct answer's letter to be written to the answer key file.

After you run the program, your *capitalsquiz1.txt* file should look something like this. Of course, your questions and answer options will depend on the outcome of your `random.shuffle()` calls:

```
Name:

Date:

Period:

                    State Capitals Quiz (Form 1)

1. What is the capital of West Virginia?
    A. Hartford
    B. Santa Fe
    C. Harrisburg
    D. Charleston

2. What is the capital of Colorado?
    A. Raleigh
    B. Harrisburg
    C. Denver
    D. Lincoln

--snip--
```

The corresponding *capitalsquiz_answers1.txt* text file will look like this:

```
1. D
2. C

--snip--
```

Randomly ordering the question set and corresponding answer key by hand would take hours to do, but with a little bit of programming knowledge, you can automate this boring task for not just a state capitals quiz but any multiple-choice exam.

Summary

Operating systems organize files into folders (also called directories), and use paths to describes their locations. Every program running on your

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

computer has a current working directory, which allows you to specify filepaths relative to the current location instead of entering the full (or absolute) path. The `pathlib` and `os.path` modules have many functions for manipulating filepaths.

Your programs can also directly interact with the contents of text files. The `open()` function can open these files to read in their contents as one large string (with the `read()` method) or as a list of strings (with the `readlines()` method). The `open()` function can also open files in write or append mode to create new text files or add to existing text files, respectively.

In previous chapters, you used the clipboard as a way of getting large amounts of text into a program, rather than typing it directly. Now you can have your programs read files from the hard drive, which is a big improvement, as files are much less volatile than the clipboard.

In the next chapter, you will learn how to handle the files themselves by copying them, deleting them, renaming them, moving them, and more.

Practice Questions

1. What is a relative path relative to?
2. What does an absolute path start with?
3. What does `Path('C:/Users') / 'Al'` evaluate to on Windows?
4. What does `'C:/Users' / 'Al'` evaluate to on Windows?
5. What do the `os.getcwd()` and `os.chdir()` functions do?
6. What are the . and .. folders?
7. In *C:\bacon\eggs\spam.txt*, which part is the directory name, and which part is the base name?
8. What three “mode” arguments can you pass to the `open()` function for plaintext files?
9. What happens if an existing file is opened in write mode?
10. What is the difference between the `read()` and `readlines()` methods?
11. What data structure does a shelf value resemble?

Practice Programs

For practice, design and write the following programs.

Mad Libs

Create a Mad Libs program that reads in text files and lets the user add their own text anywhere the word *ADJECTIVE*, *NOUN*, *ADVERB*, or *VERB* appears in the text file. For example, a text file may look like this:

```
The ADJECTIVE panda walked to the NOUN and then VERB. A nearby NOUN was
unaffected by these events.
```

The program would find these occurrences and prompt the user to replace them:

```
Enter an adjective:
silly
Enter a noun:
chandelier
Enter a verb:
screamed
Enter a noun:
pickup truck
```

It would then create the following text file:

```
The silly panda walked to the chandelier and then screamed. A nearby
pickup truck was unaffected by these events.
```

The program should print the results to the screen in addition to saving them to a new text file.

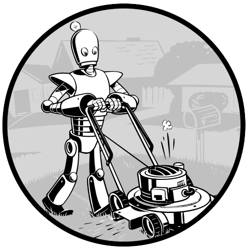
Regex Search

Write a program that opens all *.txt* files in a folder and searches for any line that matches a user-supplied regular expression, then prints the results to the screen.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

11

ORGANIZING FILES



In addition to creating and writing to new files, your programs can organize preexisting files on the hard drive. Maybe you've had the experience of going through a folder full of dozens, hundreds, or even thousands of files and copying, renaming, moving, or compressing them all by hand. Or consider tasks such as these:

- Making copies of all PDF files (and *only* the PDF files) in every subfolder of a folder
- Removing the leading zeros in the filenames for every file in a folder of hundreds of files named *spam001.txt*, *spam002.txt*, *spam003.txt*, and so on
- Compressing the contents of several folders into one ZIP file (which could serve as a simple backup system)

All this boring stuff is just begging to be automated in Python. By programming your computer to do these tasks, you can transform it into a quick-working file clerk that never makes mistakes.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

While Windows uses backslashes (\) to separate folders in a filepath, the Python code in this chapter will use forward slashes (/) instead, as they work on all operating systems.

The shutil Module

The `shutil` module has functions to let you copy, move, rename, and delete files in your Python programs. (The module's name is short for shell utilities, where *shell* is another term for a terminal command line.) To use the `shutil` functions, you'll first need to run `import shutil`.

To create an example file and folder to work with, run the following code before the interactive shell examples in this chapter:

```
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam').mkdir(exist_ok=True)
>>> with open(h / 'spam/file1.txt', 'w', encoding='utf-8') as file:
...     file.write('Hello')
...
```

This will create a folder named *spam* with a text file named *file1.txt*. The examples in this chapter will copy, move, rename, and delete this file and folder. All `shutil` functions can take filepath arguments that are either strings or `Path` objects.

Copying Files and Folders

The `shutil` module provides functions for copying files, as well as entire folders. Calling `shutil.copy(`*`source, destination`*`)` will copy the file at the path *`source`* to the folder at the path *`destination`*. Both *`source`* and *`destination`* can be strings or `Path` objects. If *`destination`* is a filename, it will be used as the new name of the copied file. If *`destination`* is a folder, the file will be copied to that folder with its original name. This function returns the path of the copied file.

Enter the following into the interactive shell to see how `shutil.copy()` works:

```
>>> import shutil
>>> from pathlib import Path
>>> h = Path.home()
❶ >>> shutil.copy(h / 'spam/file1.txt', h)
'C:\\Users\\Al\\file1.txt'
❷ >>> shutil.copy(h / 'spam/file1.txt', h / 'spam/file2.txt')
WindowsPath('C:/Users/Al/spam/file2.txt')
```

The first `shutil.copy()` call copies the file at *C:\Users\Al\spam\file1.txt* to the home folder *C:\Users\Al*. The return value is the path of the newly copied file. Note that since we specified a folder as the destination ❶, the new, copied file will have the same filename as the original *file1.txt* file.

The second `shutil.copy()` call ❷ copies the file at *C:\Users\Al\spam\file1.txt* to the *C:\Users\Al\spam* folder but gives the copied file the name *file2.txt*.

While `shutil.copy()` will copy a single file, calling `shutil.copytree(`*`source`*`, `*`destination`*`)` will copy the folder at the path *`source`*, along with all of its files and subfolders, to the folder at the path *`destination`*. The function returns the path of the copied folder.

Enter the following into the interactive shell:

```
>>> import shutil
>>> from pathlib import Path
>>> h = Path.home()
>>> shutil.copytree(h / 'spam', h / 'spam_backup')
WindowsPath('C:/Users/Al/spam_backup')
```

The `shutil.copytree()` call creates a new folder named *spam_backup* with the same content as the original *spam* folder. You have now safely backed up your precious, precious spam.

Moving and Renaming Files and Folders

Calling `shutil.move(`*`source, destination`*`)` will move the file or folder at the path *`source`* to the path *`destination`* and return a string of the new location's absolute path.

If *`destination`* points to a folder, the *`source`* file gets moved into *`destination`* and keeps its current filename. For example, enter the following into the interactive shell:

```
>>> import shutil
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam2').mkdir()
>>> shutil.move(h / 'spam/file1.txt', h / 'spam2')
'C:\\Users\\Al\\spam2\\file1.txt'
```

After creating the *spam2* folder in the home folder, this `shutil.move()` call says, "Move *C:\Users\Al\spam\file1.txt* into the folder *C:\Users\Al\spam2*." If there had been a *file1.txt* file already in *C:\Users\Al\spam2*, Python would have raised an error.

If the *`destination`* path is not an existing folder, `shutil.move()` will use this path to rename the file. In the following example, the *`source`* file is moved *and* renamed:

```
>>> shutil.move(h / 'spam/file1.txt', h / 'spam2/new_name.txt')
'C:\\Users\\Al\\spam2\\new_name.txt'
```

This line says, "Move *C:\Users\Al\spam\file1.txt* into the folder *C:\Users\Al\spam2*, and while you're at it, rename that *file1.txt* file to *new_name.txt*."

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Permanently Deleting Files and Folders

You can delete a single file or a single empty folder with functions in the `os` module, whereas to delete a folder and all of its contents, you use the `shutil` module:

- Calling `shutil.rmtree(`*`path`*`)` will delete (that is, remove) the entire folder tree at *`path`*, including all the files and subfolders it contains.
- Calling `os.unlink(`*`path`*`)` will delete the single file at *`path`*.
- Calling `os.rmdir(`*`path`*`)` will delete the folder at *`path`*. This folder must be empty.

Be careful when using these functions in your programs! It's often a good idea to first run your program with these calls commented out and `print()` calls added to show the files that would be deleted. This is called a *dry run*. Here is a Python program that was intended to delete files with the *.txt* file extension, but it has a typo (shown in bold) that causes it to delete *.rxt* files instead:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    os.unlink(filename)
```

If you had any important files ending with *.rxt*, they would have been accidentally, permanently deleted. Instead, you should have first run the program like this:

```
import os
from pathlib import Path
for filename in Path.home().glob('*.rxt'):
    #os.unlink(filename)
    print('Deleting', filename)
```

The `os.unlink()` call is now commented, so Python ignores it. Instead, you'll print the filename of the file that would have been deleted. Running this version of the program first will show you that you've accidentally told the program to delete *.rxt* files instead of *.txt* files.

You should also do dry runs for programs that copy, rename, or move files. Lastly, it may be a good idea to create a backup copy of the entire folder of any files your program touches, just in case you need to completely restore the original files. Once you're certain the program works as intended, delete the `print(filename)` line and uncomment the `os.unlink(filename)` line. Then, run the program again to actually delete the files.

Deleting to the Recycle Bin

Python's built-in `shutil.rmtree()` function irreversibly deletes files and folders. This makes the function dangerous to use, because a bug could delete files you didn't intend. A much better way to delete files and folders is with the third-party send2trash module. (See Appendix A for a more in-depth explanation of how to install third-party modules.)

Using the send2trash module's send2trash() function is much safer than Python's regular delete functions, because it will send folders and files to your computer's trash or recycling bin instead of permanently deleting them. If a bug in your program deletes something with send2trash that you didn't intend to delete, you can later restore it from the recycle bin.

After you have installed send2trash, enter the following into the interactive shell to send the file *file1.txt* to the recycle bin:

```
>>> import send2trash
>>> send2trash.send2trash('file1.txt')
```

In general, you should use the send2trash.send2trash() function to delete files and folders. But while sending files to the recycle bin lets you recover them later, it will not free up disk space like permanently deleting them does. Note that the send2trash() function can only send files to the recycle bin; it cannot pull files out of it.

Walking a Directory Tree

If you want to list all the files and subfolders in a folder, call the os.listdir() function and pass it a folder name:

```
>>> import os
>>> os.listdir(r'C:\Users\Al')
['.anaconda', '.android', '.cache', '.dotnet', '.eclipse', '.gitconfig',
--snip--
'__pycache__']
```

You can also get a list of Path objects in a folder by calling the iterdir() method:

```
>>> from pathlib import Path
>>> home = Path.home()
>>> list(home.iterdir())
[WindowsPath('C:/Users/Al/.anaconda'), WindowsPath('C:/Users/Al/.android'),
WindowsPath('C:/Users/Al/.cache'),
--snip--
WindowsPath('C:/Users/Al/__pycache__')]
```

Say you want to rename every file in some folder, and also every file in every subfolder of that folder. That is, you want to walk through the directory tree, accessing each file as you go. Writing a program to do this could get tricky; fortunately, Python provides the os.walk() function to handle this process for you.

Let's create a series of folders and files by running the following code in the interactive shell:

```
>>> from pathlib import Path
>>> h = Path.home()
>>> (h / 'spam').mkdir(exist_ok=True)
>>> (h / 'spam/eggs').mkdir(exist_ok=True)
>>> (h / 'spam/eggs2').mkdir(exist_ok=True)
>>> (h / 'spam/eggs/bacon').mkdir(exist_ok=True)
>>> for f in ['spam/file1.txt', 'spam/eggs/file2.txt', 'spam/eggs/file3.txt',
'spam/eggs/bacon/file4.txt']:
...     with open(h / f, 'w', encoding='utf-8') as file:
...         file.write('Hello')
...
>>> # At this point, the folders and files now exist.
```

This code will create the following folders and files in your home folder:

- The *spam* folder
- The *spam/file1.txt* file
- The *spam/eggs* folder
- The *spam/eggs/file2.txt* file
- The *spam/eggs/file3.txt* file
- The *spam/eggs2* folder
- The *spam/eggs/bacon* folder
- The *spam/eggs/bacon/file4.txt* file

Here is an example program that uses the `os.walk()` function on this tree of folders and renames each file to uppercase letters:

```
import os, shutil
from pathlib import Path
h = Path.home()

for folder_name, subfolders, filenames in os.walk(h / 'spam'):
    print('The current folder is ' + folder_name)

    for subfolder in subfolders:
        print('SUBFOLDER OF ' + folder_name + ': ' + subfolder)

    for filename in filenames:
        print('FILE INSIDE ' + folder_name + ': '+ filename)
        # Rename file to uppercase:
        p = Path(folder_name)
        shutil.move(p / filename, p / filename.upper())

    print('')
```

The `os.walk()` function gets passed a single string value: the path of a folder. You can use `os.walk()` in a `for` loop to walk a directory tree, much like how you can use the `range()` function to walk over a range of numbers.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Unlike `range()`, the `os.walk()` function will return three values on each iteration through the loop:

- A string of the current folder's name
- A list of strings of the subfolders in the current folder
- A list of strings of the files in the current folder

The *current folder* here refers to the folder accessed in the current iteration of the `for` loop. The `os.walk()` function doesn't change the current working directory of the program. Just as you can choose the variable name `i` in the code `for i in range(10):`, you can also choose the variable names for the three values listed earlier. I always use the descriptive names `folder_name`, `subfolders`, and `filenames`.

When I ran this program on my computer, it gave the following output:

```
The current folder is C:\Users\Al\spam
SUBFOLDER OF C:\Users\Al\spam: eggs
SUBFOLDER OF C:\Users\Al\spam: eggs2
FILE INSIDE C:\Users\Al\spam: file1.txt

The current folder is C:\Users\Al\spam\eggs
SUBFOLDER OF C:\Users\Al\spam\eggs: bacon
FILE INSIDE C:\Users\Al\spam\eggs: file2.txt
FILE INSIDE C:\Users\Al\spam\eggs: file3.txt

The current folder is C:\Users\Al\spam\eggs\bacon
FILE INSIDE C:\Users\Al\spam\eggs\bacon: file4.txt

The current folder is C:\Users\Al\spam\eggs2
```

Because `os.walk()` returns lists of strings for the `subfolder` and `filename` variables, you can use the return values in their own `for` loops. For instance, you can pass the folder and filename to functions like `shutil.move()`, as in the example.

Compressing Files with the zipfile Module

You may be familiar with ZIP files (with the *.zip* file extension), which can hold the compressed contents of many other files. Compressing a file reduces its size, which is useful when transferring it over the internet. And since a ZIP file can also contain multiple files and subfolders, it's a handy way to package several files into one. This single file, called an *archive file*, can then be, say, attached to an email.

Your Python programs can create or extract from ZIP files using functions in the `zipfile` module.

Creating and Adding to ZIP Files

To create your own compressed ZIP files, you must open the `ZipFile` object in write mode by passing `'w'` as the second argument. (Note the capital letters *Z* and *F* in the object name, which differs from the zipfile module name.) This process is similar to opening a text file in write mode by passing 'w' to the open() function. For the filename, you can pass either a string or a Path object.

When you pass a path to the write() method of a ZipFile object, Python will compress the file at that path and add it into the ZIP file. The write() method's first argument is a string of the filename to add. The second argument is the *compression type* parameter, which tells the computer what algorithm it should use to compress the files; you can always set this value to zipfile.ZIP_DEFLATED to specify the *deflate* compression algorithm, which works well on all types of data. If you don't pass this value, the write() method adds the file to the ZIP file with its regular, uncompressed size. Enter the following into the interactive shell:

```
>>> import zipfile
>>> with open('file1.txt', 'w', encoding='utf-8') as file_obj:
...     file_obj.write('Hello' * 10000)
...
>>> with zipfile.ZipFile('example.zip', 'w') as example_zip:
...     example_zip.write('file1.txt', compress_type=zipfile.ZIP_DEFLATED,
        compresslevel=9)
```

This code creates a text file named *file1.txt* and writes to it the 50,000-character string 'Hello' * 10000 (about 49KB). Then, it creates a new ZIP file named *example.zip* that has the compressed contents of *file1.txt* (about 213 bytes; highly repetitive data is also highly compressible). The compresslevel keyword argument (added in Python 3.7 and later) can be set to any value from 0 to 9, with 9 being the slowest but most compressed level. If you don't specify this keyword argument, the default is 6.

The zipfile.ZipFile() function opens a ZIP file in a with statement, in a manner similar to how the open() function opens files. This ensures that the close() method is automatically called when the execution leaves the with statement's block.

Keep in mind that, just as with writing to files, write mode will erase all existing contents of a ZIP file. If you want to simply add files to an existing ZIP file, pass 'a' as the second argument to zipfile.ZipFile() to open the ZIP file in *append mode*.

Reading ZIP Files

To read the contents of a ZIP file, you must first create a ZipFile object by calling the zipfile.ZipFile() function and passing the ZIP file's filename. Note that zipfile is the name of the Python module, and ZipFile() is the name of the function.

For example, enter the following into the interactive shell:

```
>>> import zipfile

>>> example_zip = zipfile.ZipFile('example.zip')
>>> example_zip.namelist()
['file1.txt']
```

```
>>> file1_info = example_zip.getinfo('file1.txt')
>>> file1_info.file_size
50000
>>> file1_info.compress_size
97
❶ >>> f'Compressed file is {round(file1_info.file_size / file1_info
    .compress_size, 2)}x smaller!'

'Compressed file is 515.46x smaller!'
>>> example_zip.close()
```

A `ZipFile` object has a `namelist()` method that returns a list of strings for all the files and folders contained in the ZIP file. These strings can be passed to the `getinfo()` `ZipFile` method to return a `ZipInfo` object about that particular file. `ZipInfo` objects have their own attributes, such as `file_size` and `compress_size`, which hold integers representing the original file size and compressed file size, respectively, in bytes. While a `ZipFile` object represents an entire archive file, a `ZipInfo` object holds useful information about a single file in the archive.

The command at ❶ calculates how efficiently *example.zip* is compressed by dividing the original file size by the compressed file size, then prints this information.

Extracting from ZIP Files

The `extractall()` method for `ZipFile` objects extracts all the files and folders from a ZIP file into the current working directory. Create a ZIP file named *example.zip* by following the instructions in "Creating and Adding to ZIP Files" on page XX, and then enter the following into the interactive shell:

```
>>> import zipfile
>>> example_zip = zipfile.ZipFile('example.zip')
❶ >>> example_zip.extractall()
>>> example_zip.close()
```

After running this code, Python will extract the contents of *example.zip* to the current working directory. Optionally, you can pass a folder name to `extractall()` to have it extract the files into a folder other than the current working directory. If the folder passed to the `extractall()` method doesn't exist, Python will create it. For instance, if you replaced the call at ❶ with `example_zip.extractall('C:\\spam)`, the code would extract the files from *example.zip* into a newly created *C:\spam* folder.

The `extract()` method for `ZipFile` objects will extract a single file from the ZIP file. Continue the interactive shell example by entering the following:

```
>>> example_zip.extract('file1.txt')
'C:\\Users\\Al\\Desktop\\file1.txt'
>>> example_zip.extract('file1.txt', 'C:\\some\\new\\folders')
'C:\\some\\new\\folders\\file1.txt'
>>> example_zip.close()
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The string you pass to `extract()` must match one of the strings in the list returned by `namelist()`. Optionally, you can pass a second argument to `extract()` to extract the file into a folder other than the current working directory. If this second argument is a folder that doesn't yet exist, Python will create the folder.

Project 5: Back Up a Folder into a ZIP File

Say you're working on a project whose files you keep in a folder named *C:\Users\Al\AlsPythonBook*. You're worried about losing your work, so you'd like to create ZIP file "snapshots" of the entire folder. You'd also like to keep different versions of these snapshots, so you want the ZIP file's filename to increment each time a new version is made; for example, *AlsPythonBook_1.zip*, *AlsPythonBook_2.zip*, *AlsPythonBook_3.zip*, and so on. You could do this by hand, but that would be rather annoying, and you might accidentally misnumber the ZIP files' names. It would be much simpler to run a program that does this boring task for you.

For this project, open a new file editor window and save it as *backup_to_zip.py*.

Step 1: Figure Out the ZIP File's Name

We'll place the code for this program into a function named `backup_to_zip()`. This will make it easy to copy and paste the function into other Python programs that need this functionality. At the end of the program, the function will be called to perform the backup. Make your program look like this:

```
# backup_to_zip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments

import zipfile, os
from pathlib import Path

def backup_to_zip(folder):
    # Back up the entire contents of "folder" into a ZIP file.
    folder = Path(folder)  # Make sure folder is a Path object, not string.

    # Figure out the ZIP filename this code should use, based on
    # what files already exist.
  ❶ number = 1
  ❷ while True:

        zip_filename = Path(folder.parts[-1] + '_' + str(number) + '.zip')
        if not zip_filename.exists():
            break
        number = number + 1

  ❸ # TODO: Create the ZIP file.
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backup_to_zip(Path.home() / 'spam')
```

First, import the zipfile and os modules. Next, define a backup_to_zip() function that takes just one parameter, folder. This parameter is a string or Path object to the folder whose contents should be backed up. The function will determine what filename to use for the ZIP file it will create. Then, it will create the file, walk the folder folder, and add each of the subfolders and files to the ZIP file. Write TODO comments for these steps in the source code to remind yourself to do them later ❸.

The first task, naming the ZIP file, uses the base name of the absolute path of folder. If the folder being backed up is *C:\Users\Al\spam*, the ZIP file's name should be *spam_N.zip*, where *N* is 1 the first time you run the program, *N* is 2 the second time, and so on.

You can determine what *N* should be by checking whether *spam_1.zip* already exists, then checking whether *spam_2.zip* already exists, and so on. Use a variable named number for *N* ❶, and keep incrementing it inside the loop that calls exists() to check whether the file exists ❷. The first nonexistent filename found will cause the loop to break, since it will have found the filename of the new ZIP.

Step 2: Create the New ZIP File

Next, let's create the ZIP file. Make your program look like the following:

```
# backup_to_zip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments

--snip--

    # Create the ZIP file.
    print(f'Creating {zip_filename}...')
    backup_zip = zipfile.ZipFile(zip_filename, 'w')

    # TODO: Walk the entire folder tree and compress the files in each folder.
    print('Done.')

backup_to_zip(Path.home() / 'spam')
```

Now that the new ZIP file's name is stored in the zip_filename variable, you can call zipfile.ZipFile() to actually create the ZIP file. Be sure to pass 'w' as the second argument to open the ZIP file in write mode. We'll also remove the TODO from the comment, as we've finished writing the code for this section.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Step 3: Walk the Directory Tree

Now you need to use the `os.walk()` function to do the work of listing every file in the folder and its subfolders. Make your program look like the following:

```
# backup_to_zip.py - Copies an entire folder and its contents into
# a ZIP file whose filename increments

--snip--

    # Walk the entire folder tree and compress the files in each folder.
  ❶ for folder_name, subfolders, filenames in os.walk(folder):
        folder_name = Path(folder_name)
        print(f'Adding files in folder {folder_name}...')

        # Add all the files in this folder to the ZIP file.
      ❷ for filename in filenames:
            print(f'Adding file {filename}...')
            backup_zip.write(folder_name / filename)
    backup_zip.close()
    print('Done.')


backup_to_zip(Path.home() / 'spam')
```

Use `os.walk()` in a `for` loop ❶. On each iteration, the function will return the iteration's current folder name, the subfolders in that folder, and the filenames in that folder. The nested `for` loop can go through each filename in the `filenames` list ❷. Each of these is added to the ZIP file.

When you run this program, it should produce output that looks something like this:

```
Creating spam_1.zip...
Adding files in spam...
Adding file file1.txt...
Done.
```

The second time you run it, it will put all the files in the *spam* folder into a ZIP file named *spam_2.zip*, and so on.

Ideas for Other Programs

You can walk a directory tree and add files to compressed ZIP archives in several other programs. For example, you could write programs that do the following:

- Walk a directory tree and archive just files with certain extensions, such as *.txt* or *.py*, and nothing else.
- Walk a directory tree and archive every file except the *.txt* and *.py* ones.
- Only archive the folders in a directory tree that use the most disk space or have been modified since the previous archive.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Summary

Even if you're an experienced computer user, you probably handle files manually with the mouse and keyboard. Modern file explorers make it easy to work with a few files. But sometimes you'll need to perform a task that would take hours using your computer's file explorer.

The os and shutil modules offer functions for copying, moving, renaming, and deleting files. When deleting files, you might want to use the send2trash module to move the files to the recycle bin or trash rather than permanently deleting them. And when writing programs that handle files, it's a good idea to do a dry run; comment out the code that does the actual copy, move, rename, or delete, and replace it with a print() call. This way, you can run the program and verify exactly what it will do.

Often, you'll need to perform these operations not only on files in one folder, but also on every subfolder in that folder, every subfolder in those subfolders, and so on. The os.walk() function handles this trek across the folders for you so that you can concentrate on what your program needs to do with the files in them.

The zipfile module gives you a way to compress and extract files in *.zip* archives through Python. Combined with the file-handling functions of os and shutil, zipfile makes it easy to package up several files from anywhere on your hard drive. These ZIP files are much easier to upload to websites or send as email attachments than many separate files.

Practice Questions

1. What is the difference between shutil.copy() and shutil.copytree()?
2. What function is used to rename files?
3. What is the difference between the delete functions in the send2trash and shutil modules?
4. ZipFile objects have a close() method just like File objects' close() method. What ZipFile method is equivalent to File objects' open() method?

Practice Programs

For practice, write programs to do the following tasks.

Selectively Copying

Write a program that walks through a folder tree and searches for files with a certain file extension (such as *.pdf* or *.jpg*). Copy these files from their current location to a new folder.

Deleting Unneeded Files

It's not uncommon for a few unneeded but humongous files or folders to take up the bulk of the space on your hard drive. If you're trying to free up

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

room on your computer, it's more effective to identify the largest unneeded files first.

Write a program that walks through a folder tree and searches for exceptionally large files or folders—say, ones that have a file size of more than 100MB. (Remember that, to get a file's size, you can use `os.path.getsize()` from the `os` module.) Print these files with their absolute path to the screen.

Renumbering Files

Write a program that finds all files with a given prefix, such as *spam001.txt*, *spam002.txt*, and so on, in a single folder and locates any gaps in the numbering (such as if there is a *spam001.txt* and a *spam003.txt* but no *spam002.txt*). Have the program rename all the later files to close this gap.

To create these example files (skipping *spam042.txt*, *spam086.txt*, and *spam103.txt*), run the following code:

```
>>> for i in range(1, 121):
...     if i not in (42, 86, 103):
...         with open(f'spam{str(i).zfill(3)}.txt', 'w') as file:
...             pass
...
```

As an added challenge, write another program that can insert gaps into numbered files (and bump up the numbers in the filenames after the gap) so that a new file can be inserted.

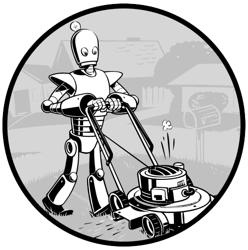
Converting Dates from American- to European-Style

Say your boss emails you thousands of files with American-style dates (MM-DD-YYYY) in their names and needs them renamed to European-style dates (DD-MM-YYYY). This boring task could take all day to do by hand! Instead, write a program that does the following:

1. Searches all filenames in the current working directory and all subdirectories for American-style dates. Use the `os.walk()` function to go through the subfolders.
2. Uses regular expressions to identify filenames with the MM-DD-YYYY pattern in them—for example, *spam12-31-1900.txt*. Assume the months and days always use two digits, and that files with non-date matches don't exist. (You won't find files named something like *99-99-9999.txt*.)
3. When a filename is found, renames the file with the month and day swapped to make it European-style. Use the `shutil.move()` function to do the renaming.

12

DESIGNING AND DEPLOYING COMMAND LINE PROGRAMS



Thus far, we've focused on running programs from Mu (or from whatever code editor you're using). This chapter discusses how to run programs from the command line terminal. The command line can be intimidating, with its cryptic commands and utter lack of user-friendly presentation, and most users stay away from it. But there are some genuine benefits to becoming familiar with it, and it's no more challenging than any of the programming you've done so far.

Once you've written a Python program to automate some task, having to open Mu every time you want to run it can be a burden. The command line is a more convenient way to execute Python scripts (especially if you share your programs with friends or co-workers who don't have Mu, or even

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Python, installed). In software development, *deployment* is the process of making our software usable outside our code editors.

A Program by Any Other Name

This chapter (and programming in general) uses a lot of terms that mean *a program* or some slight variation of the term. You could accurately call all of the following items a program. But there are subtle differences between what these names mean:

A *program* is a complete piece of software, large or small, with instructions that a computer carries out.

A *script* is a program that an interpreter runs from its source-code form rather than from a compiled, machine-code form. This is a very loose term. Python programs are often called scripts even though Python code can be compiled like other languages (as you'll learn in "Compiling Python Programs with PyInstaller" on page XX).

A *command* is a program that is often run from a text-based terminal and doesn't have a graphical user interface (GUI). All configuration is done up front by specifying command line arguments before running the command (although *interactive commands* may sometimes interrupt their operation with an "Are you sure? Y/N" question for the user). Both `dir` and `ls`, explained in "The cd, pwd, dir, and ls Commands" on page XX, are examples of commands.

A *shell script* is a single text file that conveniently runs several bundled terminal commands in one batch. This way, a user can run one shell script instead of manually entering several commands individually. On macOS and Linux, shell script files have a *.sh* file extension (or no extension), while Windows uses the term *batch file* for shell scripts with a *.bat* file extension.

An *application* is a program that has a GUI and contains multiple related features. Excel and Firefox are examples of applications. Applications usually have several files that an *installer* program sets up on your computer (and that an *uninstaller* program can remove), rather than consisting of just a single executable file copied to a computer. *App* is a common name for mobile phone and tablet applications, but the term can be used for desktop applications as well. A *web app* is a program that runs on a web server, and which users interact with over the internet through a web browser.

You're welcome to nitpick about precise definitions; these explanations should merely give you a general sense of the terms' usage. If you want to become familiar with more terminology, my book *Beyond the Basic Stuff with Python* (No Starch Press, 2021) has additional definitions in its "Programming Jargon" chapter.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Using the Terminal

Until the 1990s, when Apple and Microsoft popularized computers with GUIs that could run multiple programs simultaneously, programs were launched from a *command line interface* (CLI, pronounced either as "see-el-eye" or as a word that rhymes with "fly") and were often limited to text-based input and output. You might also hear CLIs called the *command prompt*, *terminal*, *shell*, or *console*. Software developers still make use of CLIs and often have several terminal windows open on their computers at any given time. While a text-based terminal might not have the icons, buttons, and graphics of a GUI, it's an effective way to use a computer once you've learned several commands.

To open a terminal window, do the following:

- On Windows, click the **Start** button (or press the **Windows key**) and enter **Command Prompt** (or **PowerShell** or **Terminal** if you have them installed).
- On macOS, click the Spotlight icon in the upper-right corner (or press **⌘-spacebar**) and enter **Terminal**.
- On Ubuntu Linux, press the **Windows key** to bring up Dash, and enter **Terminal**. Alternatively, use the keyboard shortcut CTRL-ALT-**T**.

Just as the interactive shell has a `>>>` prompt, the terminal will display a prompt for you to enter commands. On Windows, this will be the full path of the folder you are currently in, followed by an angle bracket (`>`):

```
C:\Users\al>your commands go here
```

On macOS, the prompt shows your username, your computer's name, and the current working directory (with your home folder represented as ~ for short), followed by a percent sign (`%`):

```
al@Als-MacBook-Pro ~ % your commands go here
```

On Ubuntu Linux, the prompt is similar to the prompt in macOS, except it begins with the username and an `@` sign:

```
al@al-VirtualBox:~$ your commands go here
```

While it's easier to run programs from the Start menu (Windows) or Spotlight (macOS), it's also possible to start them from the terminal. The Python interpreter itself is a program often run from the terminal.

In this chapter, we'll assume the Python program you want to run is named *yourScript.py* and that it's located in a *Scripts* folder under your home folder. You don't need to open Mu to access the Python interactive shell. From a terminal window, you can enter `python` on Windows or `python3` on macOS and Linux to start it. (You should then see its familiar `>>>` prompt.) To run one of your *.py* Python files from the terminal, enter its filepath after `python` or `python3`—either in its absolute form, like `python C:\Users\al\Scripts\yourScript.py`, or in its relative form, like `python yourScript.py`, if the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

current working directory is set to *C:\Users\al\Scripts*, the same folder that *yourScript.py* is in.

The cd, pwd, dir, and ls Commands

Just as all running programs have a current working directory (CWD) to which relative file paths are attached, the terminal too has a current working directory. You can see this CWD as part of the terminal prompt, or view it by running `pwd` (for *print working directory*) on macOS and Linux or the `cd` command, without any command line arguments, on Windows.

Your Python programs can change the CWD by calling the `os.chdir()` function. In the terminal, you can do the same thing by entering the `cd` command followed by the relative or absolute filepath of the folder to change to:

```
C:\Users\al>cd Desktop
C:\Users\al\Desktop>cd ..
C:\Users\al>cd C:\Windows\System32
C:\Windows\System32>
```

On Windows, you may have the added step of needing to switch the drive letter. You can't change the drive you are in with the `cd` command. Instead, enter the drive letter followed by a colon, then use `cd` to change directories on the drive:

```
C:\Windows\System32>D:
D:\>cd backup
D:\backup>
```

The `dir` command on Windows and the `ls` command on macOS and Linux will list the file and subfolder contents of the CWD:

```
C:\Users\al>dir
--snip--
08/26/2036  06:42 PM           171,304 _recursive-centaur.png
08/18/2035  11:25 AM             1,278 _viminfo
08/13/2035  12:58 AM    <DIR>          __pycache__
              77 File(s)     83,805,114 bytes
             108 Dir(s)  149,225,267,200 bytes free
```

While navigating the file system in the terminal, you'll often bounce between `cd` to change directories and `dir`/`ls` to see the contents of the directory. You can list all the executable files in the CWD by running `dir *.exe` on Windows and `file * | grep executable` on macOS and Linux. Once you are in the folder containing a program, you can run it in the following ways:

- On Windows, enter the program name with or without the *.exe* extension: `example.exe`.
- On macOS and Linux, enter `./` followed by the program name: `./example`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Of course, you can always enter the full absolute path of a program: `C:\full\path\to\example.exe` or `/full/path/to/example`.

If you want to open a non-program file, such as a text file named *example.txt*, you can open it with its associated application by entering `example.txt` on Windows or `open example.txt` on macOS and Linux. This does, from the terminal, the same thing as double-clicking the *example.txt* file's icon in a GUI. If there is no associated application set for *.txt* files, the operating system will prompt the user to select one and remember it for the future.

The PATH Environment Variable

All running programs, no matter the language they're written in, have a set of string variables called *environment variables*. One of these is the `PATH` environment variable, which contains a list of folders the terminal checks when you enter the name of a program. For example, if you enter `python` on Windows or `python3` on macOS and Linux, the terminal checks for a program with that name in the folders listed in `PATH`. Operating systems have slightly different rules for how they use `PATH`:

- Windows first checks the CWD for a program of that name, then the folders in `PATH`.
- Linux and macOS check only the folders in `PATH` and don't check the CWD at all. If you want to run a program named *example* in the CWD, you must enter `./example` rather than `example`.

To view the contents of the `PATH` environment variable, run `echo %PATH%` on Windows or `echo $PATH` on macOS and Linux. The value stored in `PATH` is a long string composed of folder names separated by a semicolon (on Windows) or a colon (on macOS and Linux). For example, on Ubuntu Linux, the `PATH` environment variable may look like the following:

```
al@al-virtual-machine:~$ echo $PATH
/home/al/.local/bin:/home/al/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/
usr/bin:/sbin:/bin:/usr/games:/usr/local/games:/snap/bin
```

If you were to enter `python3` into the Linux terminal with this `PATH`, Linux would check for a program named *python3* in the */home/al/.local/bin* folder first, then in the */home/al/bin* folder, and so on. It would eventually find *python3* in */usr/bin* and run that. The `PATH` environment variable is convenient because you can just drop a program in a folder on the `PATH` to spare yourself from `cd`-ing to its folder every time you want to run it.

Note that the terminal window doesn't search subfolders under a `PATH` folder. If *C:\Users\al\Scripts* is listed in the `PATH`, running *spam.exe* will run a *C:\Users\al\Scripts\spam.exe* file but not a *C:\Users\al\Scripts\eggs\spam.exe* file.

PATH Editing

So far, you may have been saving your *.py* files to the *mu_code* folder that the Mu editor uses by default. However, I recommend creating a *Scripts* folder under your home folder. If your username happens to be *al*, this folder would be:

- *C:\Users\al\Scripts* on Windows
- */Users/al/Scripts* on macOS
- */home/al/Scripts* on Ubuntu Linux

Let's add this folder to the `PATH`.

Windows

Windows has two sets of environment variables: system environment variables (which apply to all users) and user environment variables (which override the system environment variables but apply to the current user only). To edit them, click the **Start** menu and then enter **Edit environment variables for your account**, which should open the **Environment Variables** window.

Select **Path** from the User variable list on the top of the screen (not the System variable list on the bottom of the screen), click **Edit**, add the new folder name ***C:\Users\al\Scripts*** in the text field that appears with a semicolon separator, and click **OK**.

macOS and Linux

To add folders to the `PATH` environment variables, you'll need to edit the terminal startup script. This is the *.zshrc* file on macOS and the *.bashrc* file on Linux. Both of these files are in your home folder and contain commands that are run whenever a new terminal window is opened. On macOS, add the following to the bottom of the *.zshrc* file:

```
export PATH=/Users/al/Scripts:$PATH
```

On Linux, add the following to the bottom of the *.bashrc* file:

```
export PATH=/home/al/Scripts:$PATH
```

This line modifies `PATH` for all future terminal windows that you open, so the change won't have an effect on currently open terminal windows.

The which and where Commands

If you ever want to find out which folder in the `PATH` environment variable a program is located in, you can run the `which` program on macOS and Linux and the `where` program on Windows. For example, enter the following `which` command into the macOS terminal:

```
al@Als-MacBook-Pro ~ % which python3
/Library/Frameworks/Python.framework/Versions/3.8/bin/python3
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

On Windows, enter the following where command into the terminal:

```
C:\Users\al>where python
C:\Users\al\AppData\Local\Programs\Python\Python313\python.exe
C:\Users\al\AppData\Local\Programs\Python\Python312\python.exe
```

The where command shows each folder in PATH that contains a program named *python*. The one in the topmost folder is the version that is run when you enter python. The which and where commands are helpful if you are unsure how PATH is configured and need to find the location of a particular program.

Virtual Environments

Say you have two Python programs, one that uses version 1.0 of a package and another that uses version 2.0 of that same package. Python can't have two versions of the same package installed at the same time. If version 2.0 is not backward compatible with version 1.0, you'd be uninstalling one version and reinstalling the other each time you wanted to switch programs to run.

Python's solution to this problem is *virtual environments*; separate installations of Python that have their own set of installed third-party packages. In general, each Python application you create needs its own virtual environment. But you can use one virtual environment for all your small scripts while learning to program. Python can create virtual environments with its built-in venv module. To create a virtual environment, **cd** to your *Scripts* folder and run **python -m venv .venv** (using python3 on macOS and Linux):

```
C:\Users\al>
C:\Users\al>cd Scripts
C:\Users\al\Scripts>python -m venv .venv
```

This creates the virtual environment's files in a new folder named *.venv*. You can choose any folder name you want, but *.venv* is conventional. Files and folders whose names begin with a period are hidden, though you can follow the steps in this book's Introduction to make your operating system show them by default.

When you run python or python3 from the terminal, you'll still run your original Python installation's interpreter. To use the virtual environment's Python version, you must activate it. Do so by running the *C:\Users\al\Scripts\.venv\Scripts\activate.bat* script on Windows:

```
C:\Users\al\Scripts>cd .venv\Scripts
C:\Users\al\Scripts\.venv\Scripts>activate.bat
(.venv) C:\Users\al\Scripts\.venv\Scripts>where python.exe
C:\Users\Al\Scripts\.venv\Scripts\python.exe
C:\Users\Al\AppData\Local\Programs\Python\Python313\python.exe
C:\Users\Al\AppData\Local\Programs\Python\Python312\python.exe
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Running `where python.exe` after activating the virtual environment shows that running `python` from the terminal will run the Python interpreter in the *.venv\Scripts* folder and not the system Python (discussed shortly).

The equivalent script on macOS and Linux is *~/Scripts/.venv/bin/activate*, but due to security permissions, you can't directly run it. Instead, run the command **`source activate`**:

```
al@al-virtual-machine:~/Scripts$ cd .venv/bin
al@al-virtual-machine:~/Scripts/.venv/bin$ source activate
(.venv) al@al-virtual-machine:~/Scripts/.venv/bin$ which python3
/home/al/Scripts/.venv/bin/python3
```

Activation changes the `PATH` environment variable so that `python` or `python3` runs the Python interpreter inside the *.venv* folder instead of the original one. It also changes your terminal prompt to include `(.venv)` so that you know the virtual environment is active. Running `which python3` in the activated virtual environment shows that `python3` runs the Python interpreter in the newly created *.venv/bin* folder. These changes apply to the current terminal window only; any existing or new terminal windows won't have these environment variable or prompt changes. This fresh Python installation has only the default packages, and none of the packages you may have already installed in the original Python installation. You can confirm this by running **`python -m pip list`** to list the installed packages:

```
(.venv) C:\Users\al\Scripts\.venv\Scripts>python -m pip list
Package    Version
---------- -------
pip        23.0
setuptools 65.5.0
```

The standard practice is to create a virtual environment for each Python project you're working on, since every project could have its own unique package dependencies. However, on Windows, we can be a bit more lax with the random small scripts we write in our *Scripts* folder: they can all share this one.

The macOS and Linux operating systems have their own programs that rely on the Python installation that comes with the operating system. Installing or updating packages for this original Python installation, called the *system Python*, has the slight chance of introducing incompatibilities that can cause these programs to fail. Running your own scripts with the system Python is fine; installing third-party packages to the system Python is slightly risky, but creating a virtual environment in your *Scripts* folder is a good precaution against installing incompatible packages.

Keep in mind that Mu has its own virtual environment. When you press F5 to run a script in Mu, it won't have the packages you've installed to the *Scripts\.venv* folder's virtual environment. As you advance in your programming ability, you may find it easier to have the Mu window open to edit your code while keeping a terminal window open to run it. You can quickly swap focus between windows with the ALT-TAB keyboard shortcut on Windows and Linux and ⌘-TAB on macOS.

To deactivate the virtual environment, run `deactivate.bat` (on Windows) or `source deactivate` (on macOS and Linux) in the same folder as the *activate* script. You can also simply close the terminal window and open a new one. If you want to permanently delete the virtual environment along with its installed packages, just delete the *.venv* folder and its contents.

The following sections tell you how to deploy your script after you've set up the virtual environment and added your *Scripts* folder to `PATH`.

Installing Python Packages with Pip

Python comes with a command line package manager program called *pip* (written in lowercase unless it's at the start of a sentence). Pip is a recursive acronym for *pip installs package*. While Python's standard library comes with modules such as `sys`, `random`, and `os`, there are also hundreds of thousands of third-party packages you can find on *PyPI* (pronounced *pie-pee-eye* and not *pie-pie*), the Python Package Index, at *https://pypi.org*. In Python, a *package* is a collection of Python code made available on PyPI, and a *module* is an individual *.py* file containing Python code. You install packages from PyPI that contain modules, and you import modules with an `import` statement.

While pip is a program on its own, it's easier to run it through the Python interpreter by running `python -m pip` on Windows or `python3 -m pip` on macOS and Linux, rather than running the `pip` (on Windows) or `pip3` (on macOS and Linux) program directly. This prevents errors in the rare cases where you have multiple Python installations, your `PATH` is misconfigured, and `pip`/`pip3` is installing to a different Python interpreter than the one that runs when you enter `python`/`python3`.

DON'T USE PIP WITH ANACONDA

If you've installed the Anaconda distribution of Python instead of the regular distribution from *https://python.org*, you should avoid using pip in your conda environments. Instead, use the conda-specific package manager through the `conda` command.

To install a package from PyPI, enter the following into the terminal:

```
C:\Users\al>python -m pip install package_name
```

Remember to run `python3` if on macOS or Linux instead of `python` for these various commands. Also note that you'll need to run this from the terminal window, and not from the Python interactive shell.

To list all the packages you have installed along with their version numbers, run **`python -m pip list`**:

```
C:\Users\al>python -m pip list
Package                  Version     Editable project location
------------------------ ----------- -------------------------
altgraph                 0.17.3
argon2-cffi              21.3.0
argon2-cffi-bindings     21.2.0
async-generator          1.10
--snip--
wsproto                  1.2.0
```

You can also upgrade a package to the latest version on PyPI by running `python -m pip install -U` *`package_name`*, or install a particular version (say, 1.17.4) by running `python -m pip install` *`package_name`*`==1.17.4`.

To uninstall a package, run `python -m pip uninstall` *`package_name`*. You can find more information about pip by running `python -m pip --help`.

INSTALLING THE AUTOMATE THE BORING STUFF PACKAGE

This book covers several third-party packages. Over time, the authors of these packages may make updates that are not backward compatible with the versions used in this book. To ensure that the information in this book is accurate, please install the specific versions listed in Appendix A. Installing the latest versions will sometimes work fine, but you'll have to check the package's online documentation to learn about the new changes.

The easiest way to install all the correct versions of all the packages featured in this book is to install the automateboringstuff3 package to a virtual environment. This package acts as a container for all the packages (and matching versions) featured in this book. From a Windows terminal, activate a virtual environment and run:

```
python -m pip install automateboringstuff3
```

On macOS and Linux, activate a virtual environment and run:

```
python3 -m pip install automateboringstuff3
```

Self-Aware Python Programs

Python's standard library doesn't come with any modules that give your programs sentience. (Yet.) But several built-in variables can give your Python program useful information about itself, the operating system it's on, and the Python interpreter running it. The Python interpreter sets these variables automatically.

The `__file__` variable contains the *.py* file's path as a string. For example, if I run a *yourScript.py* file in my home folder, it evaluates to `'C:\Users\al\yourScript.py'`. Importing `from pathlib import Path` and calling `Path(__file__)` returns a `Path` object of this file. This information is useful if you need to locate files that exist in the Python program's folder. The `__file__` variable doesn't exist when you run the Python interactive shell.

The `sys.executable` variable contains the full path and file of the Python interpreter program itself, and the `sys.version` variable contains the string that appears at the top of the interactive shell with version information about the Python interpreter.

The `sys.version_info.major` and `sys.version_info.minor` variables contain integers of the major and minor version numbers of the Python interpreter. On my laptop running Python version 3.12.0 alpha 5, these are `3` and `12`, respectively. You can also pass `sys.version_info` to the `list()` function to obtain more specific information: `list(sys.version_info)` returns `[3, 12, 0, 'final', 0]` on my laptop. Having the version information in this form is much easier to work with than trying to pull it out of the `sys.version` string.

The `os.name` variable contains the string `'nt'` if running on Windows and `'posix'` if running on macOS or Linux. This is useful if your Python script needs to run different code depending on what operating system it's running on.

For more specific operating system identification, the `sys.platform` variable contains `'win32'` on Windows, `'darwin'` on macOS, and `'linux'` on Linux.

If you need highly specific information about the OS version and type of CPU, the built-in `platform` module can retrieve this information. This module is documented online at *https://docs.python.org/3/library/platform.html.*

If you need to check whether a module is installed, put the `import` statement in a `try` block and catch the `ModuleNotFoundError` exception:

```
try:
    import nonexistentModule
except ModuleNotFoundError:
    print('This code runs if nonexistentModule was not found.')
```

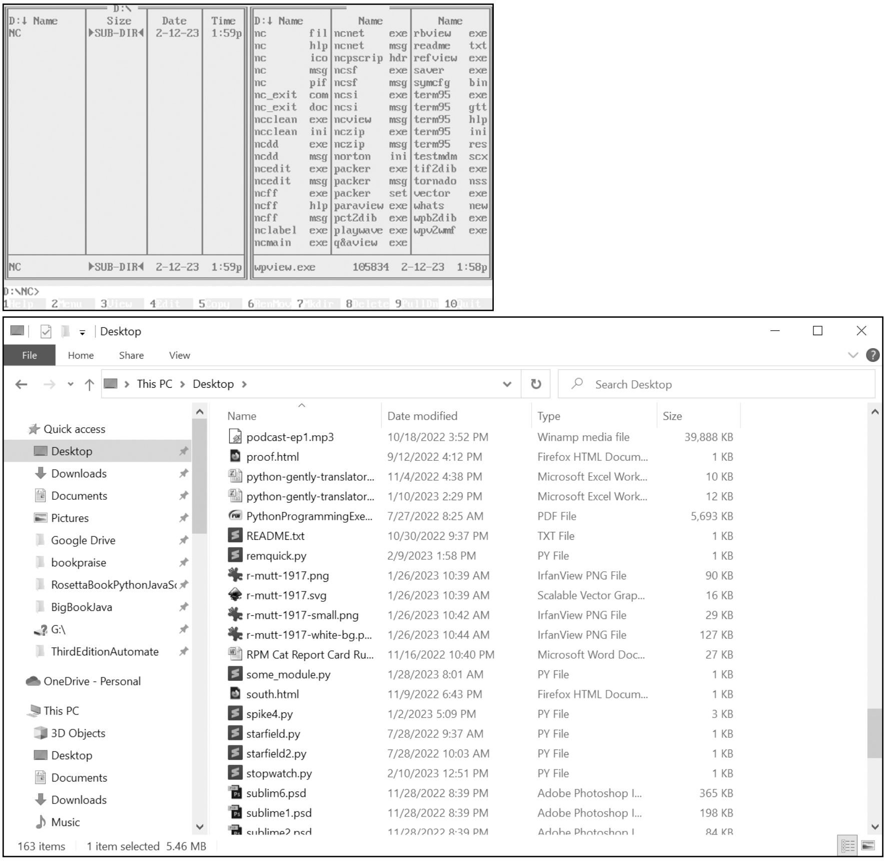
If the module is necessary for your program to function, you can put a descriptive error message here and call `sys.exit()` to terminate the program. This will be more helpful to the user than a generic error message and traceback.

Text-Based Program Design

Before GUI-supporting operating systems were common, all programs used text to communicate with their user. This book focuses on creating small, useful programs rather than professional software applications, so the programs in this book use `print()` and `input()` through a command line interface rather than the windows, buttons, and graphics that a GUI provides.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Even when limited to text, however, software applications can still provide a user interface similar to modern GUIs. Figure 12-1 shows Norton Commander, an application for browsing the file system. These kinds of applications are retroactively called *TUI* (pronounced "two-ee"), or *text-based user interface*, applications.



```
                       D:\
D:↓ Name          Size      Date     Time  D:↓ Name       Name           Name
NC             ▶SUB-DIR◀  2-12-23   1:59p  nc       fil  ncnet     exe  rbview    exe
                                           nc       hlp  ncnet     msg  readme    txt
                                           nc       ico  ncpscrip  hdr  refview   exe
                                           nc       msg  ncsf      exe  saver     exe
                                           nc       pif  ncsf      msg  symcfg    bin
                                           nc_exit  com  ncsi      exe  term95    exe
                                           nc_exit  doc  ncsi      msg  term95    gtt
                                           ncclean  exe  ncview    msg  term95    hlp
                                           ncclean  ini  nczip     exe  term95    ini
                                           ncdd     exe  nczip     msg  term95    res
                                           ncdd     msg  norton    ini  testmdm   scx
                                           ncedit   exe  packer    exe  tif2dib   exe
                                           ncedit   msg  packer    msg  tornado   nss
                                           ncff     exe  packer    set  vector    exe
                                           ncff     hlp  paraview  exe  whats     new
                                           ncff     msg  pct2dib   exe  wpb2dib   exe
                                           nclabel  exe  playwave  exe  wpv2wmf   exe
                                           ncmain   exe  q&aview   exe

NC             ▶SUB-DIR◀  2-12-23   1:59p  wpview.exe      105834  2-12-23   1:58p
D:\NC>
1Help  2Menu  3View  4Edit  5Copy  6RenMov 7Mkdir  8Delete 9PullDn 10Quit
```

Name	Date modified	Type	Size
podcast-ep1.mp3	10/18/2022 3:52 PM	Winamp media file	39,888 KB
proof.html	9/12/2022 4:12 PM	Firefox HTML Docum...	1 KB
python-gently-translator...	11/4/2022 4:38 PM	Microsoft Excel Work...	10 KB
python-gently-translator...	1/10/2023 2:29 PM	Microsoft Excel Work...	12 KB
PythonProgrammingExe...	7/27/2022 8:25 AM	PDF File	5,693 KB
README.txt	10/30/2022 9:37 PM	TXT File	1 KB
remquick.py	2/9/2023 1:58 PM	PY File	1 KB
r-mutt-1917.png	1/26/2023 10:39 AM	IrfanView PNG File	90 KB
r-mutt-1917.svg	1/26/2023 10:39 AM	Scalable Vector Grap...	16 KB
r-mutt-1917-small.png	1/26/2023 10:42 AM	IrfanView PNG File	29 KB
r-mutt-1917-white-bg.p...	1/26/2023 10:44 AM	IrfanView PNG File	127 KB
RPM Cat Report Card Ru...	11/16/2022 10:40 PM	Microsoft Word Doc...	27 KB
some_module.py	1/28/2023 8:01 AM	PY File	1 KB
south.html	11/9/2022 6:43 PM	Firefox HTML Docum...	1 KB
spike4.py	1/2/2023 5:09 PM	PY File	3 KB
starfield.py	7/28/2022 9:37 AM	PY File	1 KB
starfield2.py	7/28/2022 10:03 AM	PY File	1 KB
stopwatch.py	2/10/2023 12:51 PM	PY File	1 KB
sublim6.psd	11/28/2022 8:39 PM	Adobe Photoshop I...	365 KB
sublime1.psd	11/28/2022 8:39 PM	Adobe Photoshop I...	198 KB
sublime2.psd	11/28/2022 8:39 PM	Adobe Photoshop I	84 KB

163 items 1 item selected 5.46 MB

Figure 12-1: The text-based Norton Commander (left) alongside a modern GUI application (right)

Even if you aren't a professional software developer, the advantage of text-based user interfaces is their simplicity. This section describes several design approaches your programs can take for their user interface.

Short Command Names

Users often run text-based programs run from the command line rather than by clicking an icon on the desktop or Start menu. These commands can sometimes seem difficult to understand. When I started learning the Linux operating system, I was surprised to find that the Windows `copy` command I knew well was named `cp` on Linux. The name *copy* was much more

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

readable than *cp*. Was a terse, cryptic name really worth saving two characters' worth of typing?

As I gained more experience on the command line, I realized the answer is a firm yes. We read source code more often than we write it, so using verbose names for variables and functions helps. But we type commands into the command line more often than we read them, so in this case, the opposite is true: short command names make the command line easier to use, and they reduce strain on your wrists.

If your program is a command that you'll likely type a dozen times a day, try to think of a short name for it. You can use the `which` and `where` commands to check if the name already exists for another program. You can also do an internet search for any existing commands by that name. A short name goes a long way toward making it easy to use.

Command Line Arguments

To run a program from the command line, simply enter its name. For *.py* Python source code files, you must run the `python` (Windows) or `python3` (macOS and Linux) program and then supply the *.py* filename after it, like this: `python yourScript.py`.

The bit of text supplied after a command is called a *command line argument*. Command line arguments are passed to commands in much the same way as arguments to a function call. For example, the `ls` command by itself lists the files in the CWD. But you could also run `ls exampleFolder`, and the `exampleFolder` command line argument would direct the `ls` command to list the files in the *exampleFolder* folder. Command line arguments allow you to configure the behavior of the command.

Python scripts can access the command line scripts given to the Python interpreter from the `sys.argv` list. For example, if you entered `python3 yourScript.py hello world`, the `python3` program would receive the command line arguments and forward them to your Python script in the `sys.argv` variable. The `sys.argv` variable would contain `['yourScript.py', 'hello', 'world']`.

Note that the first item in `sys.argv` is the filename of the Python script. The remaining arguments are split by spaces. If you need to include space characters in a command line argument, put them inside double quotation marks when running the command. For example, `python3 yourScript.py "hello world"` would set `sys.argv` to `['yourScript.py', 'hello world']`.

The main usefulness of command line arguments is that you can specify a wide variety of configurations before you start the program. There's no need to go through a configuration menu or multistep process. Unfortunately, this approach means that command line arguments can become incredibly complicated and unreadable. If you pass `/?` after any Windows command or `--help` after any macOS or Linux command, you'll often find page after page of command line argument documentation.

If the set of command line arguments your program accepts is simple, then it's easiest to have your program read the `sys.argv` list directly. However, as you add more command line arguments, the possible combinations can become cumbersome to manage. Should `python yourScript.`

py spam eggs do the same thing as python yourScript.py eggs spam? If the user can have either a cheese argument or a bacon argument, what happens if they provide both? This complexity would require you to write a lot of code to handle the various edge cases. At this point, you're probably better off using Python's built-in argparse module to handle these complicated situations. The argparse module is beyond the scope of this book, but you can read its documentation online at *https://docs.python.org/3/library/argparse.html.*

Clipboard I/O

You don't need to rely on input() to read text from files or the keyboard. You can also use the clipboard for your Python program's text input and output. The cross-platform pyperclip module has a copy() function for placing text on the clipboard and a paste() function that returns the clipboard's text as a string. Pyperclip is a third-party package installed from the terminal with pip: python -m pip install pyperclip. On Linux, you'll also have to run sudo apt-get install xclip to make Pyperclip work. See Appendix A for full instructions.

All of your clipboard I/O programs will follow this basic design:

1. Import the pyperclip module.
2. Call pyperclip.paste() to obtain the input text from the clipboard.
3. Perform some work on the text.
4. Copy the results to the clipboard by passing them to pyperclip.copy().

The "Add Bullets to Wiki Markup" project in Chapter 8 is an example of this kind of program. The design of this program becomes especially useful once you've deployed it following the instructions in "Deploying Python Programs" on page XX. Just highlight the input text, press CTRL-C to copy it, and run the program. The results will be on the clipboard, ready to paste wherever needed.

Later in this chapter we'll look at two projects, the ccwd command and the Clipboard Recorder, that make use of the clipboard.

Colorful Text with Bext

You can print colorful text using the third-party Bext package built on top of Jonathan Hartley's Colorama package. Install Bext with pip by following the instructions in Appendix A. Bext only works in programs run from a terminal window, and not from Mu or most other code editors. To have print() produce colorful text, call the fg() and bg() functions to change the (foreground) text color or the background color with a string argument such as 'black', 'red', 'green', 'yellow', 'blue', 'magenta', 'purple', 'cyan', or 'white'. You can also pass 'reset' to change the color back to the terminal window's default color. For example, enter the following into the interactive shell:

```
>>> import bext
>>> bext.fg('red')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> print('This text is red.')
This text is red.
>>> bext.bg('blue')
>>> print('Red text on blue background is an ugly color scheme.')
Red text on blue background is an ugly color scheme.
>>> bext.fg('reset')
>>> bext.bg('reset')
>>> print('The text is normal again. Ah, much better.')
The text is normal again. Ah, much better.
```

Keep in mind that the user may have their terminal window set to light mode or dark mode, so there's no telling if the terminal's default appearance is black text on a white background or white text on a black background. You should also be limited in your use of color: too much can make your program look tacky or unreadable.

Bext also has some limited TUI-like features, including the following:

`bext.clear()` Clears the screen

`bext.width()` and `bext.height()` Returns the current width (in columns) and height (in rows) of the terminal window, respectively

`bext.hide()` and `bext.show()` Hides and shows the cursor, respectively

`bext.title(text)` Changes the title bar of the terminal window to the text string

`bext.goto(x, y)` Moves the cursor to column x and row y in the terminal, where 0, 0 is the top-left position

`bext.get_key()` Waits for the user to press any key and then returns a string describing the key

Think of the `bext.get_key()` function as a single-key version of `input()`. The returned string includes `'a'`, `'z'`, and `'5'`, but also keys like `'left'`, `'f1'`, and `'esc'`. The TAB and ENTER keys return `'\t'` and `'\n'`, respectively. Call `bext.get_key()` in the interactive shell to test various keys and see their return values.

For a demonstration of what Bext can do, run the source code for the ASCII Art Fish Tank program from *https://inventwithpython.com/projects/fishtank.py*. First, this program uses `bext.clear()` to clear the terminal window of all text. Next, the program calls `bext.goto()` to position the cursor and `bext.fg()` to change the text color before printing various fish out of text characters like `><)))*>`. This program is featured in my book *The Big Book of Small Python Projects* (No Starch Press, 2021).

Terminal Clearing

The `bext.clear()` function is useful if you'd like your program to remove any text left over from before it ran. You can also use it to do flipbook-style animation: call `clear()` to clear the terminal, then use `print()` calls to fill it with text, pause for a moment with Python's `time.sleep()`, and then repeat. There is a Python *one-liner* (a single line of code to do a special trick) to clear the screen that you can place in your own `clear()` function:

```
import os
def clear():
    os.system('cls' if os.name == 'nt' else 'clear')
```

This code lets your program clear the terminal screen without requiring the installation of the Bext package and only works in Python scripts run from the terminal, not from Mu or another code editor. The `os.system()` call, which you'll learn more about in Chapter 11, runs the `cls` program (on Windows) or the `clear` program (on macOS and Linux). The odd syntax here is an example of Python's *conditional expressions* (also called *ternary operators* in other languages). The syntax is *`value1`* `if` *`condition`* `else` *`value2`*, which evaluates to *`value1`* if *`condition`* is `True` and *`value2`* if *`condition`* is `False`. In our case, the conditional expression evaluates to `'cls'` if the condition `os.name == 'nt'` is `True`; otherwise, it evaluates to `'clear'`. Conditional expressions (and one-liners in general) often produce unreadable code and are usually best avoided, but this is a simple enough case.

Sound and Text Notification

Terminal programs existed before the rich audio that today's computers provide. Today, there's no reason your text-based programs must be silent. There are, however, good reasons to keep sounds to a minimum or to exclude them altogether. Sounds can provide notification that a task is complete or a problem has occurred when the user is busy looking at other windows. But, like colorful text, it's easy to overuse sounds to the point of annoyance. The user may already be doing a task that involves playing audio, or perhaps they're in an online meeting that the sound would rudely interrupt. And if the user's computer is muted, they won't hear the sound notification anyway.

If you just need to play a simple audio file, you can use the playsound third-party package. Once it's installed, you can play an audio file by calling the `playsound` module's `playsound()` function and passing the filepath of an MP3 or WAV audio file. Download the *hello.mp3 file* from *https://autbor.com/hello.mp3* (or use your own file) and enter the following into the interactive shell:

```
>>> import playsound
>>> playsound.playsound('hello.mp3')
```

The `playsound()` function won't return until the audio file has finished playing; that is, the function *blocks* until the audio has finished. Keep in mind that this will halt your program for a while if you give it a long audio file to play. If `playsound()` raises exceptions (which happens if the filename contains odd characters such as an equal sign), try passing a `Path` object of the audio file instead of a string.

Similarly, you may want to limit the text produced by your program. Under the Unix philosophy of command design, piping the text output of one command to another is easier if the command outputs only relevant

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

information, as extraneous text output would have to be filtered. Many commands keep their text output to a minimum, or they have none at all and communicate success or error with the exit code. (Exit codes are covered in Chapter 19.) However, if you aren't piping the output to another command but, as a human user, want to see more information, many commands accept a `-v` or `--verbose` command line argument to enable this *verbose mode*. Other commands take the opposite approach and flood the output with information, but offer a `-q` or `--quiet` command line argument to offer a *quiet mode* of no text output instead. (This could double as a way to mute sound notifications as well.) Or better yet, make silence the default behavior and have `--verbose` or `--beep` enable sound or alert beeps.

If your program doesn't require this level of sophistication, you can ignore this consideration. However, once you start sharing your programs with others who may use it in clever ways you didn't foresee, offering these options goes a long way toward making your programs user-friendly.

A Short Program: Snowstorm

Let's create a text-based snowstorm animation. Our program uses block text characters that fill out the top half-block, bottom half-block, and full block of a single character cell. These text characters are returned as strings by `chr(9600)`, `chr(9604)`, and `chr(9608)`, respectively, and our program stores them in the constants `TOP`, `BOTTOM`, and `FULL` to make our code more readable.

Enter the following code into a file named *snowstorm.py*:

```
import os, random, time, sys

TOP    = chr(9600)  # Character 9600 is '▀'
BOTTOM = chr(9604)  # Character 9604 is '▄'
FULL   = chr(9608)  # Character 9608 is '█'

# Set the snowstorm density to the command line argument:
DENSITY = 4  # Default snow density is 4%
if len(sys.argv) > 1:
    DENSITY = int(sys.argv[1])

def clear():
    os.system('cls' if os.name == 'nt' else 'clear')

while True:
    clear()  # Clear the terminal window.

    # Loop over each row and column:
    for y in range(20):
        for x in range(40):
            if random.randint(0, 99) < DENSITY:
                # Print snow:
                print(random.choice([TOP, BOTTOM]), end='')
            else:
                # Print empty space:
                print(' ', end='')
        print()  # Print a newline.

    # Print the snow-covered ground:
    print(FULL * 40 + '\n' + FULL * 40)
    print('(Ctrl-C to stop.)')

    time.sleep(0.2)  # Pause for a bit.
```

First, the program imports the `os`, `random`, `sys`, and `time` modules. These modules are in the Python standard library and don't require installing any third-party packages. Then, the program sets up the constants `TOP`, `BOTTOM`, and `FULL` with return values from `chr()`. The program uses these constant names because they're easier to understand than the numbers 9600, 9604, and 9608.

The user can specify the density of the snowstorm by supplying a command line argument. If no command line argument is given, then `sys.argv` is set to `['snowstorm.py']` and the program leaves `DENSITY` at `4`. But if the user were to run the program with, say, `python snowstorm.py 20`, then `sys.argv` would be set to `['snowstorm.py', '20']` and the program would update `DENSITY` to `int(sys.argv[1])`, or `20`. The user would then be able to modify the behavior of the snowstorm without having to change the source code.

Inside an infinite `while` loop, this program first clears the screen with the `cls`/`clear` one-liner. Next, it uses two nested `for` loops to loop over every row and column in a 40 × 20 space on the terminal. (You can increase or decrease these numbers to change the size of your snowstorm.) At each row and column, the program prints a single character: either a randomly selected `TOP` or `BOTTOM` character to represent snow, or an empty space character. (Only four percent of the characters are not empty spaces by default.) These `print()` calls pass the `end=''` keyword argument so that `print()` doesn't automatically print a newline character after each call. The program prints this newline itself by calling `print()` with no arguments after finishing a row.

After the nested `for` loops, the program prints two rows of 40 `FULL` characters to represent the ground, along with a reminder that the user can press CTRL-C to stop the program. All of this code produces one "frame" of the snowstorm animation, and `time.sleep(0.2)` briefly holds this frame before the execution loops back to clear the terminal and start the process all over again.

I chose the snowstorm animation because it's fun rather than practical, like a terminal-based snow globe. A more useful application of this technique is creating a *dashboard* app: a program that runs in a terminal window you leave open to convey information at a glance. This program prints relevant information, then clears the screen and reprints updated information every second, minute, hour, or other interval.

Pop-Up Message Boxes with PyMsgBox

While designing a full GUI for your program requires learning an entire code library such as Tkinter, wxPython, or PyQt, you can add small GUI

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

message boxes to your program with the PyMsgBox package. This is a third-party package you can install by running `pip install pymsgbox` from the terminal. PyMsgBox lets you create dialog boxes using Tkinter, which comes with Python on Windows and macOS. On Ubuntu Linux, you must first install Tkinter by running `sudo apt-get install python3-tk` in the terminal. Appendix A has full instructions.

PyMsgBox has functions with names that mirror JavaScript's message box functions:

`pymsgbox.alert(text)` Displays a text message until the user clicks OK, then returns the string `'OK'`

`pymsgbox.confirm(text)` Displays a text message until the user clicks OK or Cancel, then returns `'OK'` or `'Cancel'`

`pymsgbox.prompt(text)` Displays a text message along with a text field, then returns the text the user entered as a string or `None` if they clicked Cancel

`pymsgbox.password(text)` Is the same as `pymsgbox.prompt()`, but the text the user enters is masked by asterisks

These functions won't return until the user clicks OK, Cancel, or X (close). If your program requires only the occasional notification or user input, using PyMsgBox's dialog boxes could be a suitable replacement for `print()` and `input()`.

Deploying Python Programs

Once your Python program is finished, you might not want to run Mu, load the *.py* file, and then click the Run button each time you want to execute it. This section explains how to deploy your Python program so that you can run it in as few keystrokes as possible.

Be sure to follow the steps in "The PATH Environment Variable" on page XX to add a *Scripts* folder to your `PATH` environment variable. Since my username is *al*, the path to this folder is *C:\Users\al\Scripts* on Windows, */Users/al/Scripts* on macOS, and */home/al/Scripts* on Linux. Also, you'll need to set up a virtual environment for your Python scripts, as described next.

Windows

On Windows, you can bring up the Run dialog box by pressing the Windows key and the R key simultaneously (or right-clicking the Start menu button and selecting Run). This opens a small window that acts like a one-use terminal: in it, you can run a single command. To run your Python scripts from here, you'll need to do the following:

1. Place the *yourScript.py* Python script in your *Scripts* folder.
2. Create a *yourScript.bat* batch file in your *Scripts* folder to run the Python script.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Batch files contain terminal commands that can be run together by running the batch file. They have a *.bat* file extension and are similar to shell scripts on Linux or *.command* scripts on macOS. If you place a batch file named *yourScript.bat* in a PATH folder, you can run it from the Run dialog box by entering yourScript. On Windows, you don't need to enter the file extension to run *.bat* or *.exe* files.

The content of a batch file is plaintext, just like a *.py* file, so you can create batch files with Mu or a text editor like Notepad. Batch files contain one command per line. To run a *yourScript.py* file located in a folder *C:\Users\al\Scripts*, create a file named *yourScript.bat* with the following contents:

```
@call %HOMEDRIVE%%HOMEPATH%\Scripts\.venv\Scripts\activate.bat
@python %HOMEDRIVE%%HOMEPATH%\Scripts\yourScript.py %*
@pause
@deactivate
```

The batch file can be named anything, but it's easier to remember if it has the same name as the Python script. This batch file runs three commands. The first command activates the virtual environment you created for your *Scripts* folder. The @ symbol at the start makes the command itself not appear in the terminal window. The %HOMEDRIVE% environment variable is 'C:' and the %HOMEPATH% environment variable is the path to your home folder, like '\Users\al'. (The tilde [~] doesn't represent the home folder on Windows.) When combined, this provides the path to the virtual environment activation script no matter what your username is. (This is helpful if you share these files with a co-worker for them to run on their computer.) Note that the call is necessary; if a batch file (like *yourScript.bat*) runs another batch file (like *activate.bat*) without call, the rest of the first batch file's commands won't be run.

Next, the batch file runs *python.exe*, which then runs *yourScript.py*. The %* causes the batch file to forward any command line arguments it received to your Python program. It's a good idea to always include the %* in case you later add command line arguments to your Python program.

The third command runs the pause command, which causes Windows to display Press any key to continue and wait for the user to press a key. This prevents the terminal window from immediately closing after the Python program finishes so that you can see any remaining printed output. If your program has no printed output, you can omit this line. Finally, the @deactivate line deactivates the virtual environment in case you ran this batch file from the terminal and the terminal window remains open after finishing the Python program.

With the batch file set up, now you can run your Python script by pressing the Windows key + R key combination to bring up the Run dialog box and entering yourScript (followed by any command line arguments) to run the *yourScript.bat* script. Or, if you have a terminal window open, you can just enter yourScript in the terminal from any folder. This is far quicker than running it from a code editor like Mu.

If you create other Python scripts, you can reuse this batch file. Just copy the file with a new name and change the *yourScript.py* filename to the new Python script's name. Everything else can stay the same.

macOS

Pressing the COMMAND key and the spacebar simultaneously on macOS brings up Spotlight, allowing you to enter the name of a program to run. To add your own Python scripts to Spotlight, you must do the following:

1. Place the *yourScript.py* Python script in your *Scripts* folder.
2. Create a text file named *yourScript.command* to run the Python script.
3. Run `chmod u+x yourScript.command` to add execute permissions to the *yourScript.command* file.

Once you have your *.py* Python script in your *Scripts* folder, such as */Users/al/Scripts*, create a text file named *yourScript.command* in the *Scripts* folder with the following content:

```
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/yourScript.py
deactivate
```

The ~ represents the home folder, such as */Users/al*. The first line activates the virtual environment and the second line runs the Python script using the virtual environment's Python installation.

Finally, in a terminal, `cd` to *~/Scripts* and run the command `chmod u+x yourScript.command`. This adds execute permissions so that you can run the script from Spotlight. Now you'll be able to quickly run the Python script by pressing ζ-spacebar and entering *yourScript.command*. (Spotlight should autocomplete the full name for you after you enter the first few characters.) You'll also be able to run your Python script from the terminal by entering `yourScript.command`.

The *yourScript.command* file is required because if you try to run the *yourScript.py* file from Spotlight, Spotlight sees the *.py* file extension and assumes you want to open this file in Mu or some other code editor, rather than running it directly.

Note that if you use macOS's TextEdit to create the *yourScript.command* file, be sure to make it a plaintext file by pressing SHIFT-ζ-T (or clicking the Format and Make Plain Text menu items). TextEdit will also try to be helpful by automatically capitalizing `python3` to `Python3`, which causes an error in Spotlight.

Unfortunately, Spotlight has no means of letting the user pass command line arguments to the Python script. Any command line arguments must be preemptively written in the *.command* file.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Ubuntu Linux

Ubuntu Linux Dash can be brought up by pressing the Windows key and entering the name of the program you want to run. To add your Python script to Dash, you must do the following:

1. Place the *yourScript.py* Python script in your *Scripts* folder.
2. Create a shell script named *yourScript* to activate the virtual environment and run your Python script.
3. Run the `chmod u+x yourScript` command to add execute permissions to the shell script.
4. Create a *yourScript.desktop* file in the *~/.local/share/applications* folder to run the shell script from Dash.

Once you have your *.py* Python script in your *Scripts* folder, such as */home/al/Scripts*, create a text file named *yourScript* (with no file extension) in the *Scripts* folder with the following content:

```
#!/usr/bin/env bash
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/yourScript.py
read -p "Press any key to continue..." -n1 -s
deactivate
```

The ~ represents the home folder, such as */Users/al*. The first line identifies this file as a shell script. A *.sh* file extension isn't necessary for this file.

The second line activates the virtual environment, and the third line runs the Python script using the virtual environment's Python installation. The `read` command causes the terminal to display `Press any key to continue` and wait for the user to press a key. This prevents the terminal window from immediately closing after the Python program finishes so that you can see any remaining printed output. If your program has no printed output, you can omit this line.

After creating this shell script, `cd` to *~/Scripts* and run the command `chmod u+x yourScript`. This adds execute permissions so that you can run it. At this point, you'll also be able to run your Python script from the terminal by entering `yourScript`. To run your Python script from Dash, you must create a *yourScript.desktop* file.

In Mu or another text editor such as gedit, create a *yourScript.desktop* file with the following content:

```
[Desktop Entry]
Name=yourScript
Exec=gnome-terminal -- /home/al/Scripts/yourScript
Type=Application
```

Save this file to the */home/al/.local/share/applications* folder (replacing *al* with your own username) as *yourScript.desktop*. (Note that the `Exec` field requires you to spell out `/home/al`; you cannot replace the value with a ~ in

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

this file.) If your text editor doesn't show the *.local* folder (because folders that begin with a period are considered hidden), press CTRL-H in the Save File dialog box to show hidden files.

Now you'll be able to quickly run the Python script by pressing the Windows key to bring up Dash and entering *yourScript*. Dash should autocomplete the full name for you. The *yourScript* text in the `Name` field of *yourScript .desktop* will appear in Dash and can be anything, but it's convenient to give it the same name as *yourScript.py*.

Next, let's create two programs using the principles in this chapter and deploy them for easy use.

A Short Program: Copying the Current Working Directory

While the `pwd` command on macOS and Linux will print the current working directory, it's sometimes useful to copy this value to the clipboard for pasting elsewhere. For example, on Windows, I constantly find myself copying the current working directory from a terminal so that I can paste it in a Save File dialog box to save a file in that same directory. Although I could use the mouse to select the current working directory from the Windows prompt to copy (or, on macOS and Linux, run the `pwd` command to print the working directory and select that text to copy), this requires several steps for what could be a one-step process.

My idea is to write a program named *ccwd* (for *copy current working directory*). First, I'll enter `where ccwd` on Windows and `which ccwd` on macOS and Linux to make sure there isn't currently a command with the same name, then maybe also do a quick internet search for *ccwd* to be sure. The name *ccwd* is short enough to type but also unique.

As an additional feature, say the terminal's current working directory was set to *C:\Users\al\Scripts*, but what I wanted copied to the clipboard was *C:\Users\al*. I could just run the `cd ..` command, then `ccwd`, then `cd Scripts` to return to *C:\Users\al\Scripts*. But it would be easier if I could pass a relative filepath to `ccwd` as a command line argument. For example, `ccwd ..` would copy *C:\Users\al* to the clipboard when the current working directory was *C:\Users\al\Scripts*. You don't *have* to specify this command line argument (the program defaults to the current working directory if none is given), but it's available as a feature if the user wants it. These tiny improvements may seem trivial, but online retailers have "one-click buying" features on their websites because they know that slight improvements to convenience can have a major effect.

We'll use the Pyperclip package to handle the clipboard, so be sure to install this into the *Scripts* folder's virtual environment. Create a new file in Mu and enter the following content:

```
import pyperclip, os, sys
if len(sys.argv) > 1:
    os.chdir(sys.argv[1])
pyperclip.copy(os.getcwd())
```

Save this program as *ccwd.py* in the *Scripts* folder under your home folder.

The first line imports the modules the program needs. The second line checks if there are any command line arguments passed to the program. Remember that `sys.argv` is a list that always contains at least one string: the `'ccwd.py'` name of the script. If it has more than one string, we know the user supplied command line arguments to the program. In that case, the third line changes the current working directory of the program. Note that every program has its own current working directory setting, and changing this setting with `os.chdir()` doesn't change the current working directory of the terminal that ran the program. Finally, the fourth line copies the current working directory to the clipboard.

The program is finished, but to run it from a terminal, we'd have to enter the full path to it: something like `python C:\Users\al\Scripts\ccwd.py`. That's a lot to type and defeats the purpose of having a quick and easy script to copy the current working directory to the clipboard. To improve this process, let's go over the deployment steps for this program on each operating system.

Windows

Save the Python file as *C:\Users\al\Scripts\ccwd.py* (changing *al* to your username). In the same *Scripts* folder, create a *ccwd.bat* file with the following content:

```
@call %HOMEDRIVE%%HOMEPATH%\Scripts\.venv\Scripts\activate.bat
@python %HOMEDRIVE%%HOMEPATH%\Scripts\ccwd.py %*
@deactivate
```

This batch file doesn't have the `@pause` line, because it has no `print()` output. You can now run this program from the terminal in any folder by running **`ccwd`**:

```
C:\Users\al>ccwd
C:\Users\al>
```

At this point, `'C:\Users\al'` is on the clipboard.

macOS

Save the Python file as */Users/al/Scripts/ccwd.py* (changing *al* to your username). In the same *Scripts* folder, create a text file with the name *ccwd.command* and the following content:

```
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/ccwd.py
deactivate
```

Then, in a terminal, **`cd`** to the *Scripts* folder and run **`chmod u+x ccwd.command`**:

```
al@Als-MacBook-Pro ~ % cd ~/Scripts
al@Als-MacBook-Pro Scripts % chmod u+x ccwd.command
```

You can now run this program from the terminal in any folder by running **ccwd.command**:

```
al@Als-MacBook-Pro ~ % ccwd.command
al@Als-MacBook-Pro ~ %
```

At this point, `'/Users/al'` should be on the clipboard.

Ubuntu Linux

Save the Python file as */home/al/Scripts/ccwd.py* (changing *al* to your username). In the same *Scripts* folder, create a text file named *ccwd* with the following content:

```
#!/usr/bin/env bash
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/ccwd.py
deactivate
```

This *ccwd* shell script doesn't need the `read -p "Press any key to continue..." -n1 -s` line, because it will be run only from the terminal and not from Dash, and the terminal window won't disappear after running the Python script.

Then, in a terminal, **cd** to the *Scripts* folder and run **chmod u+x ccwd**:

```
al@al-VirtualBox:~$ cd ~/Scripts
al@al-VirtualBox:~/Scripts$ chmod u+x ccwd
```

You can now run this program from the terminal in any folder by running **ccwd**:

```
al@al-VirtualBox:~$ ccwd
al@al-VirtualBox:~$
```

At this point, `'/home/al'` should be on the clipboard.

A Short Program: Clipboard Recorder

Let's say that part of your job is to copy the URLs for links on a web page and paste them into a spreadsheet. (In Chapter 13, you'll learn how to scrape all the links for the HTML source of the page. But let's say you only need to copy some of them, and a human must decide which ones on a case-by-case basis.) You could follow these steps:

1. Right-click a link in a web browser.
2. Select the Copy Link or Copy Link Address item from the context menu.
3. Switch to the spreadsheet app.
4. Press CTRL-V to paste the link.
5. Switch back to the web browser.

This is a boring task, especially if the page has dozens or hundreds of links. Let's create a small clipboard-recording program to make it faster. We'll deploy this program on our computer so that we can conveniently run it when needed. Our program will monitor the clipboard to see if new text has been copied to it, and if so, it will print it to the terminal screen. This way, we can convert our five-step process into a two-step process:

1. Right-click a link in a web browser.
2. Select the Copy Link or Copy Link Address item from the context menu.

Then, the user can just copy all the text from the clipboard recorder's terminal window and paste it into the spreadsheet at once. Enter the following into the file editor and save it as *cliprec.py*:

```
import pyperclip, time

print('Recording clipboard... (Ctrl-C to stop)')
previous_content = ''
try:
    while True:
        content = pyperclip.paste()  # Get clipboard contents.

        if content != previous_content:
            # If it's different from the previous, print it:
            print(content)
            previous_content = content

        time.sleep(0.01)  # Pause to avoid hogging the CPU.
except KeyboardInterrupt:
    pass
```

Let's look at each part of this program, starting with the beginning:

```
import pyperclip, time

print('Recording clipboard... (Ctrl-C to stop)')
previous_content = ''
```

This program copies and pastes text from the clipboard, so we'll need to import the pyperclip module. We'll also use the time module's sleep() function to keep the program from hogging the CPU. The program displays a quick message to say that it is running and reminds the user that CTRL-C causes the program to stop. The program will know that the clipboard contents have changed by keeping track of what the contents were previously with a previous_content variable, which is initially set to a blank string.

```
try:
    while True:
        content = pyperclip.paste()  # Get clipboard contents.
```

The bulk of the program exists in an infinite while loop, which itself is inside a try block. When the user presses CTRL-C, Python raises a

KeyboardInterrupt exception, causing the execution to move to the except block at the bottom of the source code.

This loop continuously monitors the contents of the clipboard and notes each time the user copies new text to it. Within the loop, the first step is to collect the text on the clipboard by calling pyperclip.paste().

```
        if content != previous_content:
            # If it's different from the previous, print it:
            print(content)
            previous_content = content
```

If the current content on the clipboard is different from the previous content, then the program prints the current content and updates previous_content to content. This sets up the loop for the next time the user copies new text to the clipboard.

```
        time.sleep(0.01)  # Pause to avoid hogging the CPU.
```

If the clipboard contents are the same as the previously obtained clipboard contents, we could have our program do nothing. However, this program can easily execute this loop tens of thousands of times a second, and it's unlikely that the user will be updating the clipboard that frequently. (They'd probably wear out the CTRL and C keys on their keyboard if they tried.) To prevent the program from hogging the CPU by running this unproductive loop as fast as possible, we introduce a 0.01-second delay so that the loop checks for clipboard updates a mere 100 times a second.

```
except KeyboardInterrupt:
    pass
```

The last part of the program is the except clause, which uses Python's pass statement. This statement literally does nothing, but Python is expecting at least one line in the block that follows the except statement. This is what the pass statement was created for. When the user presses CTRL-C, the execution moves to this except clause and proceeds to the end of the program, where it terminates.

With this app running, the user can copy several things without having to switch back and forth between apps. Small programs like this can make your workflow much easier, especially if you have to do this work every day. Now let's deploy this program on each operating system.

Windows

Save the Python file as *C:\Users\al\Scripts\cliprec.py* (changing *al* to your username). In the same *Scripts* folder, create a *cliprec.bat* file with the following content:

```
@call %HOMEDRIVE%%HOMEPATH%\Scripts\.venv\Scripts\activate.bat
@python %HOMEDRIVE%%HOMEPATH%\Scripts\cliprec.py %*
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
@pause
@deactivate
```

You can now run this program from the terminal or by pressing the Windows key and the R key simultaneously to open the Run dialog, and then entering `cliprec`.

macOS

Save the Python file as */Users/al/Scripts/cliprec.py* (changing *al* to your username). In the same *Scripts* folder, create a text file with the name *cliprec.command* and the following content:

```
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/cliprec.py
deactivate
```

Then, in a terminal, **`cd`** to the *Scripts* folder and run **`chmod u+x cliprec.command`**:

```
al@Als-MacBook-Pro ~ % cd ~/Scripts
al@Als-MacBook-Pro Scripts % chmod u+x cliprec.command
```

You can now run this program by pressing ζ-spacebar to bring up Spotlight and entering `cliprec.command`.

Ubuntu Linux

Save the Python file as */home/al/Scripts/cliprec.py* (changing *al* to your username). In the same *Scripts* folder, create a text file named *cliprec* with the following content:

```
#!/usr/bin/env bash
source ~/Scripts/.venv/bin/activate
python3 ~/Scripts/cliprec.py
read -p "Press any key to continue..." -n1 -s
deactivate
```

Then, in a terminal, **`cd`** to the *Scripts* folder and run **`chmod u+x cliprec`**:

```
al@al-VirtualBox:~$ cd ~/Scripts
al@al-VirtualBox:~/Scripts$ chmod u+x cliprec
```

Finally, create a text file saved as *~/.local/share/applications/cliprec.desktop* with the following content:

```
[Desktop Entry]
Name=Clipboard Recorder
Exec=gnome-terminal -- /home/al/cliprec
Type=Application
```

You can now run this program by pressing the Windows key to bring up Dash and entering `Clipboard Recorder`, or entering the first few characters of the name and letting autocomplete finish it for you.

Compiling Python Programs with PyInstaller

Python is often called an interpreted language, though programming languages themselves are neither interpreted nor compiled. You can create an interpreter or compiler for any language. Instead, programs written in Python are mostly *run* by interpreters. But it's also possible to create executable programs from Python code with the PyInstaller package, which generates executable programs you can run from the command line.

PyInstaller doesn't compile Python programs into machine code per se; rather, it creates an executable program that contains a copy of the Python interpreter and your script. As such, these programs tend to be fairly large. Even a simple "Hello, world" program compiled with PyInstaller can be close to 8MB in size, literally a thousand times larger than a version written in assembly language. However, the benefit of compiling your Python program is that you can share your program with others who don't have Python installed. You'd be able to send them one executable file.

You can install PyInstaller by running `pip install pyinstaller`. You must run PyInstaller on the operating system that you want the executable to run on. That is, if you're on Windows, PyInstaller can create a Windows executable program but not a macOS or Linux program, and vice versa.

From the terminal, run the following command (using `python3` instead of `python` on macOS and Linux) to compile a Python script named *yourScript.py*:

```
C:\Users\al>python -m PyInstaller --onefile yourScript.py
378 INFO: PyInstaller: X.X.X
378 INFO: Python: 3.XX.XX
392 INFO: Platform: Windows-XX-XX.X.XXXX
393 INFO: wrote C:\Users\al\Desktop\hello-test\hello.spec
399 INFO: UPX is not available.
--snip--
11940 INFO: Appending PKG archive to EXE
11950 INFO: Fixing EXE headers
13622 INFO: Building EXE from EXE-00.toc completed successfully.
```

Notice that you must enter *PyInstaller* with a capital *P* and capital *I* or else you'll get a "No module named pyinstaller" error message. Also, note that the `--onefile` argument has two dashes.

After running PyInstaller, there will be a *build* folder (which you can delete) and a *dist* folder. The *dist* folder contains the executable program. You don't need to create a virtual environment for it. You can then copy this program to other computers or send it as an email attachment. Keep in mind that, as a security precaution, many email providers may block emails that contain an executable program.

The instructions here work for basic Python programs. The online documentation at *https://pyinstaller.org/* contains further details.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Summary

In this chapter, you learned how to take your programs out of the code editor and deploy them so that users can run them quickly and conveniently. You also learned more guidelines about how to design programs that have text-based user interfaces instead of more modern graphical ones. While a GUI is user friendly, TUIs are simpler to code. When you need to automate tasks for yourself, making your program look like a professional application isn't worth the extra effort. You just need something that works.

That said, there are several ways you can design your program to be easy to use. Often, this involves the command line terminal, which many users aren't familiar with. It may take a while to learn command line concepts such as navigating the file system with `cd` and `dir`/`ls`, the `PATH` environment variable, and command line arguments. But the terminal allows you to very quickly issue commands and run programs, especially after you finish writing the programs and deploy them.

This chapter also covered several third-party packages. The Bext package lets you add colorful text, position the cursor, and clear the screen. The PyMsgBox package creates GUI boxes for alerts or basic input without using the terminal window. Because you may one day run programs that require incompatible versions of the same package, it's best to run scripts in separate virtual environments. You can create virtual environments with the `venv` module that comes with Python. Virtual environments are activated from the terminal, and can prevent you from breaking existing programs by giving you a separate place to install packages.

Finally, the PyInstaller package allows you to compile your *.py* files into executable programs. They may be several megabytes in size, but you can share these programs with co-workers who might not have Python (and the third-party packages your program uses) on their computer.

This chapter didn't cover many programming language concepts; rather, it covered how to make your programs usable and convenient on a day-to-day basis. At this point, you know enough Python syntax to create basic programs (although there's always more to learn!). In Part II of this book, you'll explore several third-party packages that extend the capabilities of your Python programs.

Practice Questions

1. What command lists folder contents on Windows? What about on macOS and Linux?
2. What does the `PATH` environment variable contain?
3. What does the `__file__` variable contain?
4. What command erases the text from the terminal window on Windows? What about on macOS and Linux?
5. How do you create a new virtual environment?
6. What command line argument should you pass to PyInstaller when compiling programs?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Practice Program: Make Your Programs Deployable

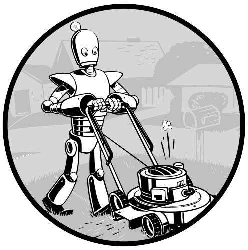
Make your existing programs easy to run by creating shell scripts in a `PATH` folder that executes them, or else compiling them with PyInstaller. Do this for the following projects:

- "Back Up a Folder into a ZIP File" from Chapter 11
- "Phone Number and Email Address Extractor" from Chapter 9
- "Add Bullets to Wiki Markup" from Chapter 8
- "Interactive Chessboard Simulator" from Chapter 7
- Any other programs you've created and want to easily launch or share with others

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

13

WEB SCRAPING



In those rare, terrifying moments when I'm without Wi-Fi, I realize just how much of what I do on the computer is really what I do on the internet. Out of sheer habit, I'll find myself trying to check email, read social media, or answer the question, "Did Kurtwood Smith have any major roles before he was in the original 1987 *RoboCop*?"[1]

Because so much work on a computer involves going on the internet, it'd be great if your programs could get online. *Web scraping* is a term for using a program to download and process content from the web. For example, Google runs many web scraping programs to index web pages for

[1] The answer is no.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

its search engine. In this chapter, you'll learn about the following modules, which make it easy to scrape web pages in Python:

webbrowser Comes with Python and opens a browser to a specific page

requests Downloads files and web pages from the internet

Beautiful Soup (bs4) Parses HTML, the format that web pages are written in, to extract the information you want

Selenium Launches and controls a web browser, such as by filling in forms and simulating mouse clicks

Playwright Launches and controls a web browser; newer than Selenium and has some additional features

HTTP and HTTPS

When you visit a website, its web address, such as *https://autbor.com/example3.html*, is known as a *uniform resource locator (URL)*. The *HTTPS* in the URL stands for *Hypertext Transfer Protocol Secure*, the protocol that your web browser uses to access websites. The packages in this chapter allow your scripts to access web servers through this protocol.

More precisely, HTTPS is an encrypted version of HTTP, so it protects your privacy while you use the internet. If you were using HTTP, identity thieves, national intelligence agencies, and your internet service provider could view the content of the web pages you visited, including any passwords and credit card information you submit. Using a virtual private network (VPN) could keep your internet service provider from viewing your internet traffic; however, now the VPN provider would be able to view your traffic. An unscrupulous VPN provider could then sell information about what websites you visit to data brokers. (Tom Scott discusses what a VPN does and does not provide in his video, "This Video Is Sponsored by VPN.")

By contrast, any web page content you view with HTTPS will be encrypted and hidden. Websites used to use HTTPS only for pages that sent passwords and credit card numbers, but nowadays, most websites encrypt all traffic. Keep in mind, though, that the identity of the website you visit can still be known; no one will be able to see exactly what you download from CatPhotos.com, but they will see that you were connecting to the CatPhotos.com website and can figure out that you were probably looking at photos of cats. The Tor Browser, which uses the Tor anonymization network, can provide true anonymous browsing, and you can download it from *https://www.torproject.org/download/*.

Project 6: Run a Program with the webbrowser Module

Let's learn about Python's `webbrowser` module by using it in a programming project. The `webbrowser` module's `open()` function can launch a new browser to a specified URL. Enter the following into the interactive shell:

```
>>> import webbrowser
>>> webbrowser.open('https://inventwithpython.com/')
```

A web browser tab will open to the URL *https://inventwithpython.com*. This is about the only thing the `webbrowser` module can do. Even so, the `open()` function does make some interesting things possible.

For example, it's tedious to copy a street address to the clipboard every time you'd like to bring up a map of it on OpenStreetMap. You could eliminate a few steps from this task by writing a simple script to automatically launch the map in your browser using the contents of your clipboard. This way, you'd only have to copy the address to a clipboard and run the script for the map to load for you. We can put the address directly into the OpenStreetMap URL, so all we need is the `webbrowser.open()` function.

This is what your program does:

- Gets a street address from the command line arguments or clipboard
- Opens the web browser to the OpenStreetMap page for that address

This means your code needs to do the following:

- Read the command line arguments from `sys.argv`.
- Read the clipboard contents.
- Call the `webbrowser.open()` function to open the web browser.
- Open a new file editor tab and save it as *showmap.py*.

Step 1: Figure Out the URL

By following the instructions in Chapter 12, set up a *showmap.py* file so that when you run it from the command line, like so . . .

```
C:\Users\al> showmap 777 Valencia St, San Francisco, CA 94110
```

. . . the script will use the command line arguments instead of the clipboard. If there are no command line arguments, then the program will know to use the contents of the clipboard.

To do so, you need to figure out what URL to use for a given street address. When you load *https://www.openstreetmap.org* in the browser and search for an address, the URL in the address bar looks something like this: *https://www.openstreetmap.org/search?query=777%20Valencia%20St%2C%20San%20Francisco%2C%20CA%2094110#map=19/37.75897/-122.42142*.

We can test that the URL doesn't need the *#map* part by taking it out of the address bar and visiting that site to confirm it still loads properly. So, your program can be set to open a web browser to *https://www.openstreetmap.org/*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

search?query=<your_address_string> (where *<your_address_string>* is the address you want to map). Note that your browser automatically handles any necessary URL encoding, such as converting space characters in the URL to *%20*.

Step 2: Handle the Command Line Arguments

Make your code look like this:

```
# showmap.py - Launches a map in the browser using an address from the
# command line or clipboard

import webbrowser, sys
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])

# TODO: Get address from clipboard.

# TODO: Open the web browser.
```

First, you need to import the `webbrowser` module for launching the browser and the sys module for reading the potential command line arguments. The `sys.argv` variable stores the program's filename and command line arguments as a list. If this list has more than just the filename in it, then `len(sys.argv)` evaluates to an integer greater than `1`, meaning that command line arguments have indeed been provided.

Command line arguments are usually separated by spaces, but in this case, you'll want to interpret all of the arguments as a single string. Because `sys.argv` is a list of strings, you can pass it to the `join()` method, which returns a single string value. You don't want the program name in this string, so you should pass `sys.argv[1:]` instead of `sys.argv` to chop off the first element of the array. The final string that this expression evaluates to is stored in the `address` variable.

If you run the program by entering this into the command line

```
showmap 777 Valencia St, San Francisco, CA 94110
```

the `sys.argv` variable will contain this list value:

```
['showmap.py', '777', 'Valencia', 'St, ', 'San', 'Francisco, ', 'CA', '94110']
```

After you've joined `sys.argv[1:]` with a space character, the `address` variable will contain the string `'777 Valencia St, San Francisco, CA 94110'`.

Step 3: Retrieve the Clipboard Content

To fetch the URL from the clipboard, make your code look like the following:

```
# showmap.py - Launches a map in the browser using an address from the
# command line or clipboard
```

```
import webbrowser, sys, pyperclip
if len(sys.argv) > 1:
    # Get address from command line.
    address = ' '.join(sys.argv[1:])
else:
    # Get address from clipboard.
    address = pyperclip.paste()

# Open the web browser.
webbrowser.open('https://www.openstreetmap.org/search?query=' + address)
```

If there are no command line arguments, the program will assume the address is stored on the clipboard. You can get the clipboard content with `pyperclip.paste()` and store it in a variable named `address`. Finally, to launch a web browser with the OpenStreetMap URL, call `webbrowser.open()`.

While some of the programs you write will perform huge tasks that save you hours, it can be just as satisfying to use a program that conveniently saves you a few seconds each time you perform a common task, such as getting a map of an address. Table 13-1 compares the steps needed to display a map with and without *showmap.py*.

Table 13-1: Getting a Map with and Without *showmap.py*

Manually getting a map	Using *showmap.py*
1. Highlight the address.	1. Highlight the address.
2. Copy the address.	2. Copy the address.
3. Open the web browser.	3. Run *showmap.py*.
4. Go to *https://www.openstreetmap.org/*.	
5. Click the address text field.	
6. Paste the address.	
7. Press ENTER.	

We're fortunate that the OpenStreetMap website doesn't require any interaction to get a map; we can just put the address information directly into the URL. The *showmap.py* script makes this task less tedious, especially if you do it frequently.

Ideas for Similar Programs

As long as you have a URL, the `webbrowser` module lets users cut out the step of opening the browser and directing themselves to a website. Other programs could use this functionality to do the following:

- Open all links on a page in separate browser tabs.
- Open the browser to the URL for your local weather site.
- Open several social networking sites or bookmarked sites that you regularly check.
- Open a local *.html* file on your hard drive.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The last suggestion is useful for displaying help files. While your program could use `print()` to display a help page to the user, calling `webbrowser.open()` to open a *.html* file with help information allows the page to have different fonts, color, tables, and images. Instead of the *https://* prefix, use the *file://* prefix. For example, your *Desktop* folder should have a local *help.html* file at *file:///C:/Users/al/Desktop/help.html* on Windows or *file:///Users/al/Desktop/help.html* on macOS.

Downloading Files from the Web with the requests Module

The `requests` module lets you easily download files from the web without having to worry about complicated issues such as network errors, connection routing, and data compression. The module doesn't come with Python, so you'll have to install it before you can use it by following the instructions in Appendix A.

Downloading Web Pages

The `requests.get()` function takes a string representing a URL to download. By calling `type()` on the function's return value, you can see that it returns a `Response` object, which contains the response that the web server gave for your request. I'll explain the `Response` object in more detail later, but for now, enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests
❶ >>> response = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> type(response)
<class 'requests.models.Response'>
❷ >>> response.status_code == requests.codes.ok
True
>>> len(response.text)
178978
>>> print(response.text[:210])
The Project Gutenberg EBook of Romeo and Juliet, by William Shakespeare

This eBook is for the use of anyone anywhere at no cost and with
almost no restrictions whatsoever.  You may copy it, give it away or
```

The URL takes you to a web page containing the entire text of *Romeo and Juliet* ❶. You can tell that the request for the web page succeeded by checking the `Response` object's `status_code` attribute. If it's equal to the value of `requests.codes.ok`, everything went fine ❷. (Incidentally, the status code for "OK" in HTTP is 200. You may already be familiar with the 404 status code for "Not Found.")

If the request succeeded, the downloaded web page is stored as a string in the `Response` object's `text` variable. This large string consists of the entire play; the call to `len(response.text)` shows you that it's more than 178,000 characters long. Finally, calling `print(response.text[:210])` displays only the first 210 characters.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If the request failed and displayed an error message, like "Failed to establish a new connection" or "Max retries exceeded," check your internet connection. Connecting to servers can be quite complicated, and I can't give a full list of possible problems here. You can find common causes of your error by doing a web search of the error message in quotes. Also keep in mind that if you download a web page with `requests`, you'll get only the HTML content of the web page. You must download images and other media separately.

Checking for Errors

As you've seen, the `Response` object has a `status_code` attribute that you can check against `requests.codes.ok` to see whether the download succeeded. A simpler way to check for success is to call the `raise_for_status()` method on the `Response` object. This method will raise an exception if an error occurred while downloading the file and will do nothing if the download succeeded. Enter the following into the interactive shell:

```
>>> response = requests.get('https://inventwithpython.com/page_that_does_not_exist')
>>> response.raise_for_status()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>

  File "C:\Users\Al\AppData\Local\Programs\Python\PythonXX\lib\site-packages\
requests\models.py", line 940, in raise_for_status
    raise HTTPError(http_error_msg, response=self)
requests.exceptions.HTTPError: 404 Client Error: Not Found for url:
https://inventwithpython.com/page_that_does_not_exist.html
```

The `raise_for_status()` method is an easy way to ensure that a program halts if a bad download occurs. Generally, you'll want your program to stop as soon as some unexpected error happens. If a failed download isn't a deal breaker, you can wrap the `raise_for_status()` line with `try` and `except` statements to handle this error case without crashing:

```
import requests
response = requests.get('https://inventwithpython.com/page_that_does_not_exist')
try:
    response.raise_for_status()
except Exception as exc:
    print(f'There was a problem: {exc}')
```

This `raise_for_status()` method call causes the program to output the following:

```
There was a problem: 404 Client Error: Not Found for url:
https://inventwithpython.com/page_that_does_not_exist.html
```

Always call `raise_for_status()` after calling `requests.get()`. You should be sure that the download has actually worked before your program continues.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Saving Downloaded Files to the Hard Drive

From here, you can save the web page to a file on your hard drive with the standard `open()` function and `write()` method. However, you must open the file in *write binary* mode by passing the string `'wb'` as the second argument to `open()`. Even if the page is in plaintext (such as the *Romeo and Juliet* text you downloaded earlier), you need to write binary data instead of text data in order to maintain the Unicode encoding of the text.

To write the web page to a file, you can use a `for` loop with the `Response` object's `iter_content()` method:

```
>>> import requests
>>> response = requests.get('https://automatetheboringstuff.com/files/rj.txt')
>>> response.raise_for_status()
>>> with open('RomeoAndJuliet.txt', 'wb') as play_file:
...     for chunk in response.iter_content(100000):
...             play_file.write(chunk)
...
100000
78978
```

The `iter_content()` method returns "chunks" of the content on each iteration through the loop. Each chunk is of the *bytes* data type, and you get to specify how many bytes each chunk will contain. One hundred thousand bytes is generally a good size, so pass `100000` as the argument to `iter_content()`.

The file *RomeoAndJuliet.txt* now exists in the current working directory. Note that while the filename on the website was *rj.txt*, the file on your hard drive has a different filename.

The `write()` method returns the number of bytes written to the file. In the previous example, there were 100,000 bytes in the first chunk, and the remaining part of the file needed only 78,978 bytes.

A REVIEW OF FILE DOWNLOADING AND SAVING

To review, here's the complete process for downloading and saving a file:

- Call `requests.get()` to download the file.
- Call `open()` with `'wb'` to create a new file in write binary mode.
- Loop over the `Response` object's `iter_content()` method.
- Call `write()` on each iteration to write the content to the file.

That's all there is to the `requests` module! You can learn about the module's other features at *https://requests.readthedocs.io/en/latest/*.

If you want to download video files from websites such as YouTube, Facebook, Twitter, or other sites, use the `yt-dlp` module instead, covered in Chapter 24.

Accessing a Weather API

The apps you use are designed to interact with human users. However, you can write programs to interact with other programs through their *application programming interface (API)*, which is the specification that defines how one piece of software (such as your Python program) can communicate with another piece of software (such as the web server for a weather site). Online services often have APIs. For example, you could write a Python script to post to your social media accounts or download new photos. In this section, we'll write a script that accesses weather information from the free OpenWeather website.

Almost all online services require you to register an email address to use their API. Even if the API is free, they may have limits to how many API requests you can make per hour or day. If you're worried about receiving spam email, you can use a temporary, disposable email address service such as *https://10minutemail.com*. Keep in mind that you should use such services only to register for online accounts you don't care about, as an unscrupulous email service could take control of your online account by making a password reset request in your name.

To start, sign up for a free account at *https://openweathermap.org*. The free account tier limits you to making 60 API requests per minute. This is more than enough for your small or medium-sized programming projects. If your program needs more than this limit (say, because it's processing requests from hundreds of simultaneous visitors to your website), you can purchase a paid account tier. Online services will give you an *API key*, which is sort of a password that identifies your account in your API requests. Keep this API key a secret! Anyone with this key can make API requests credited to your account. If you write a program that uses an API key, consider having the program read a text file that contains the key instead of including the API key directly in your source code. This way, you can share your program with others (who can sign up for their own API key) without worrying about exceeding the API request limits of your account.

Many HTTP APIs deliver their responses as one large string. This string is often formatted as JSON or XML. Chapter 18 covers JSON and XML in more detail, but for now, you just need to know that `json.loads(response.text)` returns a Python data structure of lists and dictionaries containing the JSON data in `response.text`. The examples in this chapter store this data in a variable named `response_data`, but this is an arbitrary choice, and you can use any variable name you'd like.

All online services document how to use their API. OpenWeather provides its documentation at *https://openweathermap.org/api*. After you've logged in to your account and obtained your API key from the *My API keys* page at *https://home.openweathermap.org/api_keys*, use it in the following interactive shell code. I'll use '30ee784a80d81480dab1749d33980112' as a fake API key in this example. Don't use this fake API key example in your code; it won't work.

First, you'll use OpenWeather to find the latitude and longitude of San Francisco:

```
>>> import requests
>>> city_name = 'San Francisco'
>>> state_code = 'CA'
>>> country_code = 'US'
>>> API_key = '30ee784a80d81480dab1749d33980112'  # Not a real API key
>>> response = requests.get(f'https://api.openweathermap.org/geo/1.0/
direct?q={city_name},{state_code},{country_code}&appid={API_key}')
>>> response.text  # This is a Python string.
'[{"name":"San Francisco","local_names":{"id":"San Francisco",
--snip--
,"lat":37.7790262,"lon":-122.419906,"country":"US","state":"California"}]'
>>> import json
>>> response_data = json.loads(response.text)
>>> response_data  # This is a Python data structure.
[{"name":"San Francisco","local_names":{"id":"San Francisco",
--snip--
,"lat":37.7790262,"lon":-122.419906,"country":"US","state":"California"}]
```

To understand the data in the response, you should look at the online API documentation for OpenWeather or examine the dictionary in response_data in the interactive shell. You'll learn that the response is a list whose first item (at index 0) is a dictionary with keys 'lat' and 'lon'. The values for these keys are float values of the latitude and longitude:

```
>>> response_data[0]['lat']
37.7790262
>>> response_data[0]['lon']
-122.419906
```

The specific URL used to make an API request is called the *endpoint*. The f-strings in this example replace the parts in curly brackets with the values of variables. The direct?q={city_name},{state_code},{country_code}&appid={API_key}' in the previous example becomes direct?q=San Francisco,CA,US&appid=30ee784a80d81480dab1749d33980112'.

Next, you can use this latitude and longitude information to find the current temperature of San Francisco:

```
>>> lat = json.loads(response.text)[0]['lat']
>>> lon = json.loads(response.text)[0]['lon']
>>> response = requests.get(f'https://api.openweathermap.org/data/2.5/
weather?lat={lat}&lon={lon}&appid={API_key}')
>>> response_data = json.loads(response.text)
>>> response_data
{'coord': {'lon': -122.4199, 'lat': 37.779}, 'weather': [{'id': 803,
--snip--
'timezone': -25200, 'id': 5391959, 'name': 'San Francisco', 'cod': 200}
```

```
>>> response_data['main']['temp']
285.44
>>> round(285.44 - 273.15, 1)  # Convert Kelvin to Celsius.
12.3
>>> round(285.44 * (9 / 5) - 459.67, 1)  # Convert Kelvin to Fahrenheit.
54.1
```

Notice that OpenWeather returns the temperature in Kelvin, so you'll need to do some math to get the temperature in Celsius or Fahrenheit.

Let's break down the full URL of the geolocation endpoint from the previous example:

https:// The *scheme* used to access the server, which is the protocol name (almost always HTTPS for online APIs) followed by a colon and two forward slashes.

api.openweathermap.org The domain name of the web server that handles the API request.

/geo/1.0/direct The path of the API.

?q={city_name},{state_code},{country_code}&appid={API_key} The URL's query string. The parts inside curly brackets need to be replaced by real values; you can think of them as parameters for a function call. In URL encoding, the parameter name and argument value are separated by an equal sign, and multiple parameter-argument pairs are separated by an ampersand.

You can take the endpoint URL (with the completed query string) and paste it into your web browser to view the response text directly. This is often a good practice when you're first learning how to use an API. The response text for web-based APIs is most often formatted in JSON or XML.

To avoid confusion when updating an API, most online services include a version number as part of the URL. Over time, a service may release new versions of the API and deprecate older versions. At this point, you'll have to update the code in your scripts to continue to make use of them.

The free tier of OpenWeather also provides five-day forecasts and information about precipitation, wind, and air pollution. The documentation web pages show you what URLs to access to get this data, as well as the structure of the JSON response for these API calls. The code in the next few sections assumes you've run `response_data = json.loads(response.text)` to convert the text returned from the website into a Python data structure.

Requesting a Latitude and Longitude

The endpoint to get the latitude and longitude coordinates of a city is *https://api.openweathermap.org/geo/1.0/direct?q={city_name},{state_code},{country_code}&appid={API_key}*. The state code refers to the state's abbreviation and is required only for cities in the United States. The country code is the two- or three-letter ISO 3166 code, listed at *https://en.wikipedia.org/wiki/List_of_ISO_3166_country_codes*. For example, use the code `'US'` for the United States or `'NZ'` for New Zealand. After converting the response JSON text

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data[0]['lat']` Holds the degrees latitude of the city as a float value

`response_data[0]['lon']` Holds the degrees longitude of the city as a float value

If the city name matches multiple responses, the list in `response_data` will contain different dictionaries at `response_data[0]`, `response_data[1]`, and so on. If OpenWeather is unable to locate the city, `response_data` will be an empty list.

Fetching the Current Weather

The endpoint to get current weather information based on some latitude and longitude is *https://api.openweathermap.org/data/2.5/weather?lat={lat}&lon={lon}&appid={API_key}*. After converting the response JSON text into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data['weather'][0]['main']` Holds a string description, such as `'Clear'`, `'Rain'`, or `'Snow'`

`response_data['weather'][0]['description']` Holds a more descriptive string, such as `'light rain'`, `'moderate rain'`, or `'extreme rain'`

`response_data['main']['temp']` Holds the current temperature in Kelvin

`response_data['main']['feels_like']` Holds the human perception of the temperature in Kelvin

`response_data['main']['humidity']` Holds the humidity as a percentage

If you supplied an incorrect latitude or longitude argument, `response_data` will be a dictionary, like `{"cod":"400","message":"wrong latitude"}`.

Getting a Weather Forecast

The endpoint to get a five-day forecast based on some latitude and longitude is *https://api.openweathermap.org/data/2.5/forecast?lat={lat}&lon={lon}&appid={API_key}*. After converting the response JSON text into a Python data structure in a variable named `response_data`, you can retrieve the following information:

`response_data['list']` Holds a list of dictionaries containing the weather predictions for a given time.

`response_data['list'][0]['dt']` Holds a timestamp in the form of a Unix epoch float. Pass this value as an argument to `datetime.datetime.fromtimestamp()` to obtain the timestamp as a `datetime` object. Chapter 19 discusses Python's `datetime` module in more detail.

`response_data['list'][0]['main']` Holds a dictionary with keys like `'temp'`, `'feels_like'`, `'humidity'`, and others.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

`response_data['list'][0]['weather'][0]` Holds a dictionary of descriptions with keys like `'main'`, `'description'`, and others.

The list in `response_data['list']` holds 40 dictionaries with forecasts at three-hour increments for the next five days, though this may change in future versions of the API.

Exploring APIs

Other websites, such as *https://weather.gov* and *https://weatherapi.com*, provide their own free weather APIs. Every API works differently, but they're often accessed as requests over HTTPS, in which case you can use the Requests library and return responses formatted as JSON or XML text. However, someone may have created a third-party Python package to make using these APIs easier, with Python functions that handle accessing the endpoints and parsing the response for you. You can find these packages on *https://pypi.org*; read the package documentation to learn about their use.

Understanding HTML

Before you pick apart web pages, you must learn some *Hypertext Markup Language (HTML)* basics. HTML is the format in which web pages are written, while *Cascading Style Sheets (CSS)* are a way to make categorical changes to the look of HTML elements in a web page. This chapter assumes you have some basic experience with HTML, but if you need a beginner tutorial, I suggest one of the following sites:

- *https://developer.mozilla.org/en-US/docs/Learn/HTML*
- *https://www.freecodecamp.org/news/html-coding-introduction-course-for-beginners*
- *https://www.khanacademy.org/computing/computer-programming/html-css*

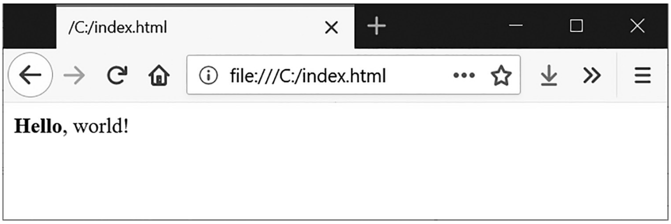
In this section, you'll also learn how to access your web browser's powerful Developer Tools, which make scraping information from the web much easier.

Exploring the Format

An HTML file is a plaintext file with the *.html* file extension. The text in these files is surrounded by HTML *tags*, which are words enclosed in angle brackets (<>). The tags tell the browser how to format the web page. A starting tag and closing tag can enclose some text to form an HTML *element*. The text to display is the content between the starting and closing tags. For example, the following HTML will display **`Hello,`** `world!` in the browser, with *Hello* in bold:

```
<b>Hello</b>, world!
```

In a browser, this HTML will look as shown in Figure 13-1.



Hello, world!

Figure 13-1: Hello, world! *rendered in the browser*

The opening `<b>` tag says that the enclosed text will appear in bold. The closing `</b>` tag tells the browser where the end of the bold text is. Together, they form an element: `<b>Hello</b>`.

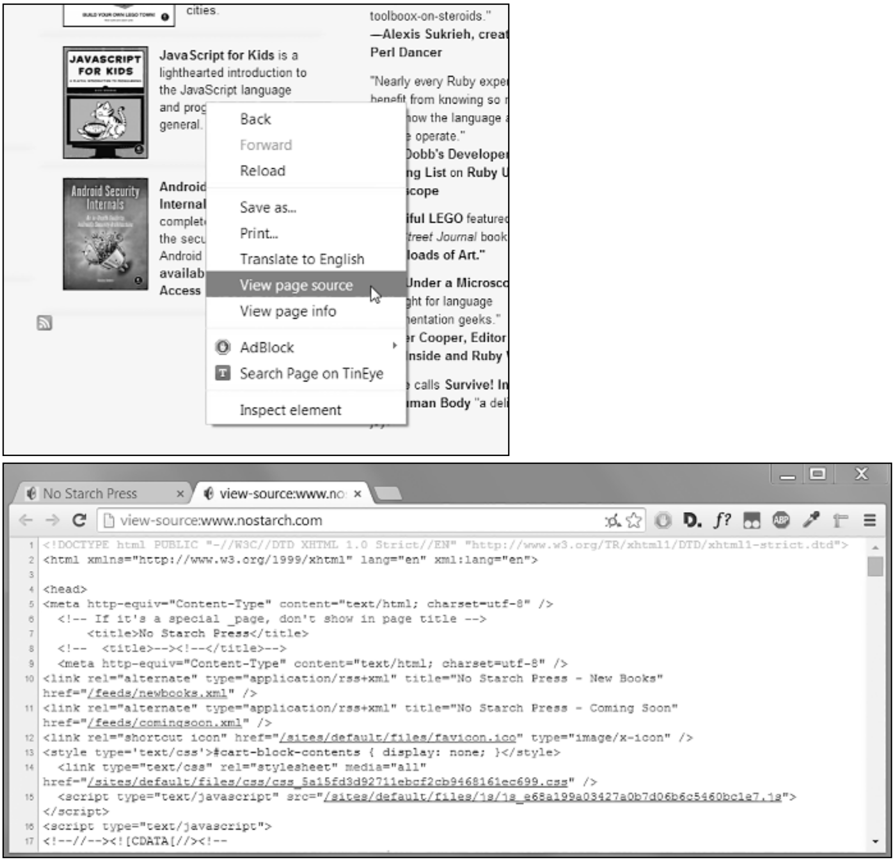
There are many different tags in HTML. Some of these tags have extra properties in the form of *attributes* within the angle brackets. For example, the `<a>` tag encloses text that should be a link, and the `href` attribute determines what URL to link to. Here's an example:

```
<a href="https://inventwithpython.com">This text is a link</a>
```

Some elements have an `id` attribute used to uniquely identify the element in the page. You'll often instruct your programs to seek out an element by its `id` attribute, so finding this attribute using the browser's Developer Tools is a common task when writing web scraping programs.

Viewing a Web Page's Source

You'll need to look at the HTML of the web pages your programs will work with, called the *source*. To do this, right-click any web page in your web browser (or CTRL-click it on macOS), and select **View Source** or **View page source** (Figure 13-2). The source is the text your browser actually receives. The browser knows how to display, or *render*, the web page from this HTML.



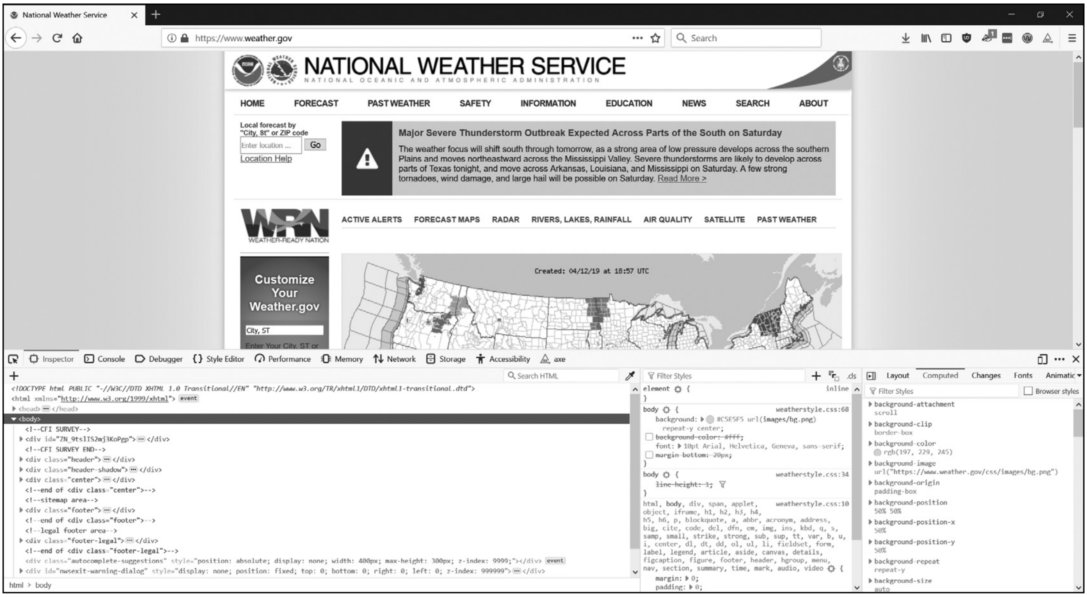
```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" lang="en" xml:lang="en">

<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <!-- If it's a special _page, don't show in page title -->
      <title>No Starch Press</title>
  <!--  <title>--><!--</title>-->
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - New Books"
href="/feeds/newbooks.xml" />
<link rel="alternate" type="application/rss+xml" title="No Starch Press - Coming Soon"
href="/feeds/comingsoon.xml" />
<link rel="shortcut icon" href="/sites/default/files/favicon.ico" type="image/x-icon" />
<style type='text/css'>#cart-block-contents { display: none; }</style>
  <link type="text/css" rel="stylesheet" media="all"
href="/sites/default/files/css/css_5a15fd3d92711ebcf2cb9468161ec699.css" />
  <script type="text/javascript" src="/sites/default/files/js/js_e68a199a03427a0b7d06b6c5460bc1e7.js">
</script>
<script type="text/javascript">
<!--//--><![CDATA[//><!--
```

Figure 13-2: Viewing the source of a web page

Go ahead and view the source HTML of some of your favorite sites. It's fine if you don't fully understand what you're seeing. You won't need HTML mastery to write simple web scraping programs. You just need enough knowledge to pick out data from an existing site.

Opening Your Browser's Developer Tools

In addition to viewing a web page's source, you can look through a page's HTML using your browser's Developer Tools. In Firefox, Chrome, and Microsoft Edge, you can press F12 to make the tools appear (Figure 13-3). Pressing F12 again will make them disappear.



Major Severe Thunderstorm Outbreak Expected Across Parts of the South on Saturday

The weather focus will shift south through tomorrow, as a strong area of low pressure develops across the southern Plains and moves northeastward across the Mississippi Valley. Severe thunderstorms are likely to develop across parts of Texas tonight, and move across Arkansas, Louisiana, and Mississippi on Saturday. A few strong tornadoes, wind damage, and large hail will be possible on Saturday. Read More >

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN" "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"> event
▶ <head> ••• </head>
▼ <body>
    <!--CFI SURVEY-->
  ▶ <div id="ZN_9tslIS2mj3KoPgp"> ••• </div>
    <!--CFI SURVEY END-->
  ▶ <div class="header"> ••• </div>
  ▶ <div class="header-shadow"> ••• </div>
  ▶ <div class="center"> ••• </div>
    <!--end of <div class="center">-->
    <!--sitemap area-->
  ▶ <div class="footer"> ••• </div>
    <!--end of <div class="footer">-->
    <!--legal footer area-->
  ▶ <div class="footer-legal"> ••• </div>
    <!--end of <div class="footer-legal">-->
    <div class="autocomplete-suggestions" style="position: absolute; display: none; width: 400px; max-height: 300px; z-index: 9999;"></div> event
  ▶ <div id="nwsexit-warning-dialog" style="display: none; position: fixed; top: 0; bottom: 0; right: 0; left: 0; z-index: 999999"> ••• </div>
```

```
element {                                        inline
}
body {                              weatherstyle.css:68
   background: ▶ #C5E5F5 url(images/bg.png)
     repeat-y center;
□  ~~background-color: #fff;~~
   font: ▶ 10pt Arial, Helvetica, Geneva, sans-serif;
□  ~~margin-bottom: 20px;~~
}
body {                              weatherstyle.css:34
   ~~line-height: 1;~~
}
html, body, div, span, applet,      weatherstyle.css:10
object, iframe, h1, h2, h3, h4,
h5, h6, p, blockquote, a, abbr, acronym, address,
big, cite, code, del, dfn, em, img, ins, kbd, q, s,
samp, small, strike, strong, sub, sup, tt, var, b, u,
i, center, dl, dt, dd, ol, ul, li, fieldset, form,
label, legend, article, aside, canvas, details,
figcaption, figure, footer, header, hgroup, menu,
nav, section, summary, time, mark, audio, video {
   margin: ▶ 0;
   padding: ▶ 0;
```

```
▶ background-attachment
  scroll
▶ background-clip
  border-box
▶ background-color
  rgb(197, 229, 245)
▶ background-image
  url("https://www.weather.gov/css/images/bg.png")
▶ background-origin
  padding-box
▶ background-position
  50% 50%
▶ background-position-x
  50%
▶ background-position-y
  50%
▶ background-repeat
  repeat-y
▶ background-size
  auto
```

Figure 13-3: The Developer Tools window in the Chrome browser

After enabling or installing the Developer Tools in your browser, right-click any part of the web page and select **Inspect Element** from the context menu to bring up the HTML responsible for that part of the page. This will help you parse HTML for your web scraping programs.

DON'T USE REGULAR EXPRESSIONS TO PARSE HTML

Locating a specific piece of HTML (or a piece of XML, JSON, TOML, or YAML) in a string seems like a perfect case for regular expressions. However, I advise you not to do this. HTML can be formatted in many ways and still be considered valid, but trying to capture all these possible variations in a regular expression is tedious and error prone. Using a module developed specifically for parsing HTML, such as bs4, is less likely to result in bugs.

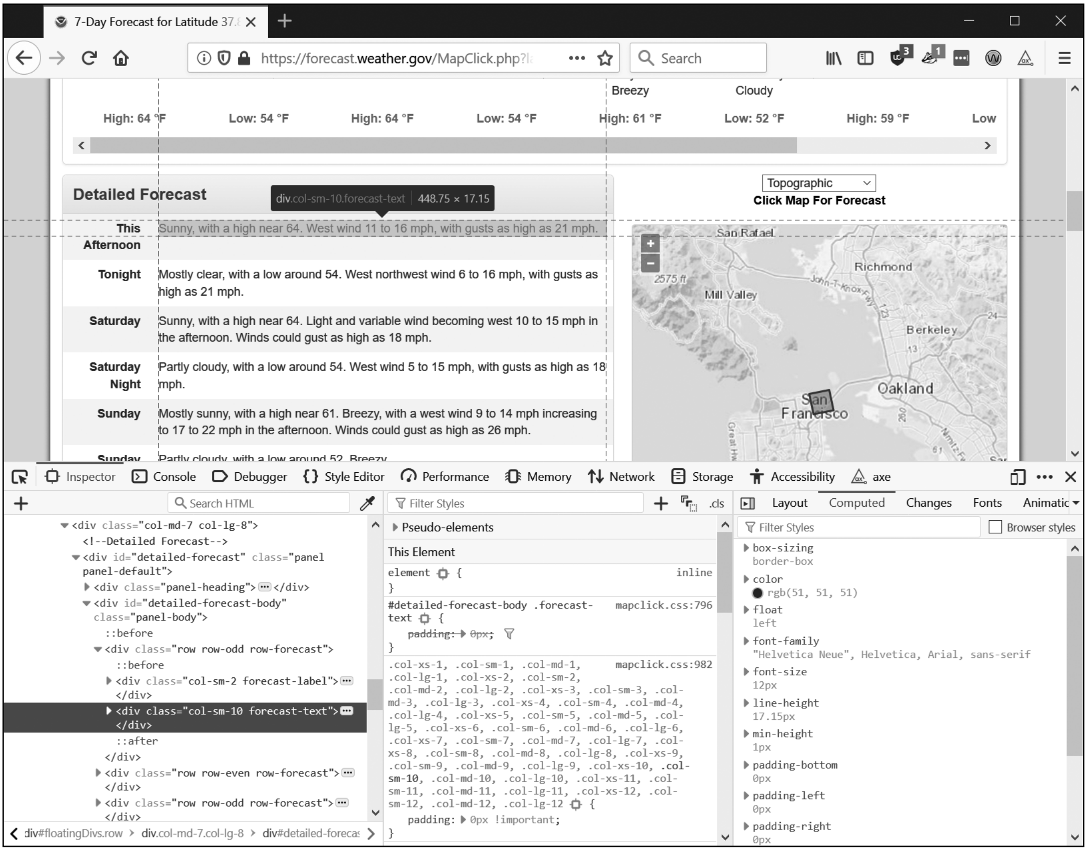
You can find an extended argument for why you shouldn't parse HTML with regular expressions at *https://stackoverflow.com/a/1732454/1893164*.

Finding HTML Elements

Once your program has downloaded a web page using the requests module, you'll have the page's HTML content as a single string value. Now you need to figure out which part of the HTML corresponds to the information on the web page you're interested in.

This is where the browser's Developer Tools can help. Say you want to write a program to pull weather forecast data from *https://weather.gov*. Before writing any code, do a little research. If you visit the site and search for the 94105 ZIP code, it should take you to a page showing the forecast for that area.

What if you're interested in scraping the weather information for that ZIP code? Right-click that information on the page (or CTRL-click on macOS) and select **Inspect Element** from the context menu that appears. This brings up the Developer Tools window, which shows you the HTML that produces that particular part of the web page. Figure 13-4 shows the Developer Tools open to the HTML of the nearest forecast. Note that if the *https://weather.gov* site changes the design of its web pages, you'll need to repeat this process to inspect the new elements.



This Afternoon Sunny, with a high near 64. West wind 11 to 16 mph, with gusts as high as 21 mph.

Tonight Mostly clear, with a low around 54. West northwest wind 6 to 16 mph, with gusts as high as 21 mph.

Saturday Sunny, with a high near 64. Light and variable wind becoming west 10 to 15 mph in the afternoon. Winds could gust as high as 18 mph.

Saturday Night Partly cloudy, with a low around 54. West wind 5 to 15 mph, with gusts as high as 18 mph.

Sunday Mostly sunny, with a high near 61. Breezy, with a west wind 9 to 14 mph increasing to 17 to 22 mph in the afternoon. Winds could gust as high as 26 mph.

Sunday Partly cloudy, with a low around 52. Breezy

```
<div class="col-md-7 col-lg-8">
  <!--Detailed Forecast-->
  <div id="detailed-forecast" class="panel
  panel-default">
   <div class="panel-heading"> ⋯ </div>
   <div id="detailed-forecast-body"
    class="panel-body">
     ::before
    <div class="row row-odd row-forecast">
      ::before
     <div class="col-sm-2 forecast-label"> ⋯
     </div>
     <div class="col-sm-10 forecast-text"> ⋯
     </div>
      ::after
    </div>
    <div class="row row-even row-forecast"> ⋯
    </div>
    <div class="row row-odd row-forecast"> ⋯
    </div>
```

```
Pseudo-elements
This Element
element {                                         inline
}
#detailed-forecast-body .forecast-    mapclick.css:796
text {
  ~~padding: 0px;~~
}
.col-xs-1, .col-sm-1, .col-md-1,      mapclick.css:982
.col-lg-1, .col-xs-2, .col-sm-2,
.col-md-2, .col-lg-2, .col-xs-3, .col-sm-3, .col-
md-3, .col-lg-3, .col-xs-4, .col-sm-4, .col-md-4,
.col-lg-4, .col-xs-5, .col-sm-5, .col-md-5, .col-
lg-5, .col-xs-6, .col-sm-6, .col-md-6, .col-lg-6,
.col-xs-7, .col-sm-7, .col-md-7, .col-lg-7, .col-
xs-8, .col-sm-8, .col-md-8, .col-lg-8, .col-xs-9,
.col-sm-9, .col-md-9, .col-lg-9, .col-xs-10, .col-
sm-10, .col-md-10, .col-lg-10, .col-xs-11, .col-
sm-11, .col-md-11, .col-lg-11, .col-xs-12, .col-
sm-12, .col-md-12, .col-lg-12 {
  padding: 0px !important;
}
```

```
box-sizing
  border-box
color
  ● rgb(51, 51, 51)
float
  left
font-family
  "Helvetica Neue", Helvetica, Arial, sans-serif
font-size
  12px
line-height
  17.15px
min-height
  1px
padding-bottom
  0px
padding-left
  0px
padding-right
  0px
```

Figure 13-4: Inspecting the element that holds forecast text

From the Developer Tools, you can see that the HTML responsible for the forecast part of the web page is this:

```
<div class="col-sm-10 forecast-text">Sunny, with a high near 64.
West wind 11 to 16 mph, with gusts as high as 21 mph.</div>
```

This is exactly what you were looking for! It seems that the forecast information is contained inside a `<div>` element with the `forecast-text` CSS class.

Right-click this element in the browser's developer console and, from the context menu that appears, select **Copy ▸ CSS Selector**. This option copies a string such as `'div.row-odd:nth-child(1) > div:nth-child(2)'` to the clipboard. You can pass it to Beautiful Soup's `select()` method or Selenium's `find_element()` method, as explained later in this chapter, to find the element in the string.

The *CSS selector* syntax used in this string specifies which HTML elements to retrieve from a web page. The full selector syntax is beyond the scope of this book, but you can obtain the selector from the browser Developer Tools, as we did here. *XPath* is another syntax for selecting HTML elements, but is also beyond the scope of this book.

Keep in mind that when a website changes its layout, you'll need to update the HTML tags your scripts check. This can happen with little or no warning, so be sure to keep an eye on your program in case it suddenly displays errors about not being able to find elements. In general, it's better to use a website's API if it offers one, as it's much less likely to change than the website itself.

Parsing HTML with Beautiful Soup

Beautiful Soup is a package for extracting information from an HTML page. You'll use the name `beautifulsoup4` to install the package but the shorter module name `bs4` to import it. In this section, we'll use Beautiful Soup to *parse* (that is, analyze and extract the parts of) the HTML file at *https://autbor.com/example3.html*, which has the following content:

```
<!-- This is an HTML comment. -->

<html>
<head>
    <title>Example Website Title</title>
    <style>
        .slogan {
            color: gray;
            font-size: 2em;
        }
    </style>
</head>
<body>
    <h1>Example Website</h1>
    <p>This &lt;p&gt; tag puts <b>content</b> into a <i>single</i> paragraph.</p>
    <p><a href="https://inventwithpython.com">This text is a link</a> to books by <span id=
"author">Al Sweigart</span>.</p>
    <p><img src="wow_such_zophie_thumb.webp" alt="Close-up of my cat Zophie." /></p>
    <p class="slogan">Learn to program in Python!</p>
    <form>
        <p><label>Username: <input id="login_user" placeholder="admin" /></label></p>
        <p><label>Password: <input id="login_pass" type="password" placeholder="swordfish" />
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    </form>
</label></p>
    <p><label>Agree to disagree: <input type="checkbox" /></label><input type="submit"
value="Fake Button" /></p>
</body>
</html>
```

Note that the login form on this page is fake and is included for cosmetic value.

Even a simple HTML file involves many different tags and attributes, and matters quickly get confusing when it comes to complex websites. Thankfully, Beautiful Soup makes working with HTML much easier.

Creating a Beautiful Soup Object

The `bs4.BeautifulSoup()` function accepts a string containing the HTML it will parse, then returns a `BeautifulSoup` object. For example, enter the following into the interactive shell while your computer is connected to the internet:

```
>>> import requests, bs4
>>> res = requests.get('https://autbor.com/example3.html')
>>> res.raise_for_status()
>>> example_soup = bs4.BeautifulSoup(res.text, 'html.parser')
>>> type(example_soup)
<class 'bs4.BeautifulSoup'>
```

This code uses `requests.get()` to download the main page of the Automate the Boring Stuff website and then passes the response's `text` attribute to `bs4.BeautifulSoup()`. Beautiful Soup can parse different formats, and the `'html.parser'` argument tells it that we are parsing HTML. Finally, the code stores the returned `BeautifulSoup` object in a variable named `example_soup`.

You can also load an HTML file from your hard drive by passing a `File` object to `bs4.BeautifulSoup()`. Enter the following into the interactive shell (after making sure the *example3.html* file is in the working directory):

```
>>> import bs4
>>> with open('example3.html') as example_file:
...     example_soup = bs4.BeautifulSoup(example_file, 'html.parser')
...
>>> type(example_soup)
<class 'bs4.BeautifulSoup'>
```

Once you have a `BeautifulSoup` object, you can use its methods to locate specific parts of an HTML document.

Finding an Element

You can retrieve a web page element from a `BeautifulSoup` object by calling its `select()` method and passing a CSS selector string for the element you're looking for. The method returns a list of `Tag` objects, which represent

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

matching HTML elements. Table 13-2 shows examples of the most common CSS selector patterns using select().

Table 13-2: Examples of CSS Selectors

Selector passed to the select() method	Will match . . .
soup.select('div')	All elements named <div>
soup.select('#author')	The element with an id attribute of author
soup.select('.notice')	All elements that use a CSS class attribute named notice
soup.select('div span')	All elements named <span> that are within an element named <div>
soup.select('div > span')	All elements named <span> that are *directly* within an element named <div>, with no other element in between
soup.select('input[name]')	All elements named <input> that have a name attribute with any value
soup.select('input[type="button"]')	All elements named <input> that have an attribute named type with the value button

You can combine the various selector patterns to make sophisticated matches. For example, soup.select('p #author') matches any element that has an id attribute of author, as long as it's also inside a <p> element.

You can pass tag values to the str() function to show the HTML tags they represent. Tag values also have an attrs attribute containing all their HTML attributes as a dictionary. For example, download the *https://autbor.com/example3.html* page as *example3.html*, then enter the following into the interactive shell:

```
>>> import bs4
>>> example_file = open('example3.html')
>>> example_soup = bs4.BeautifulSoup(example_file.read(), 'html.parser')
>>> elems = example_soup.select('#author')
>>> type(elems) # elems is a list of Tag objects.
<class 'bs4.element.ResultSet'>
>>> len(elems)
1
>>> type(elems[0])
<class 'bs4.element.Tag'>
>>> str(elems[0])  # The Tag object as a string
'<span id="author">Al Sweigart</span>'
>>> elems[0].getText()  # The inner text of the element
'Al Sweigart'
>>> elems[0].attrs
{'id': 'author'}
```

This code finds the element with id="author" in our example HTML. We use select('#author') to return a list of all the elements with id="author". We then store this list of Tag objects in the variable elems. Running len(elems) tells us there is one Tag object in the list, meaning there was one match.

Passing the element to str() returns a string with the starting and closing tags and the element's text. Calling getText() on the element returns the element's text, or the content between the opening and closing tags: in this case, 'Al Sweigart'. Finally, attrs gives us a dictionary with the element's attribute, 'id', and the value of the id attribute, 'author'.

You can also pull all the <p> elements from the BeautifulSoup object. Enter this into the interactive shell:

```
>>> p_elems = example_soup.select('p')
>>> str(p_elems[0])
'<p>This &lt;p&gt; tag puts <b>content</b> into a <i>single</i> paragraph.</p>'
>>> p_elems[0].getText()
'This <p> tag puts content into a single paragraph.'
>>> str(p_elems[1])
'<p> <a href="https://inventwithpython.com/">This text is a link</a> to books by
<span id="author">Al Sweigart</span>.</p>'
>>> p_elems[1].getText()
'This text is a link to books by Al Sweigart.'
>>> str(p_elems[2])
'<p><img alt="Close-up of my cat Zophie." src="wow_such_zophie_thumb.webp"/></p>'
>>> p_elems[2].getText()
''
```

This time, select() gives us a list of three matches, which we store in p_elems. Using str() on p_elems[0], p_elems[1], and p_elems[2] shows you each element as a string, and using getText() on each element shows you its text.

Getting Data from an Element's Attributes

The get() method for Tag objects lets you access HTML attribute values from an element. You'll pass the method an attribute name as a string and receive that attribute's value. Using *example3.html* from *https://autbor.com/example3.html*, enter the following into the interactive shell:

```
>>> import bs4
>>> soup = bs4.BeautifulSoup(open('example3.html'), 'html.parser')
>>> span_elem = soup.select('span')[0]
>>> str(span_elem)
'<span id="author">Al Sweigart</span>'
>>> span_elem.get('id')
'author'
>>> span_elem.get('some_nonexistent_addr') == None
True
>>> span_elem.attrs
{'id': 'author'}
```

Here, we use select() to find any <span> elements and then store the first matched element in span_elem. Passing the attribute name 'id' to get() returns the attribute's value, 'author'.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Project 7: Open All Search Results

When I look up a topic on a search engine, I don't look at just one search result at a time. By *middle-clicking* a search result link (or clicking it while holding CTRL), I open the first several links in a bunch of new tabs to read later. I search the internet often enough that this workflow—opening my browser, searching for a topic, and middle-clicking several links one by one—is tedious. It would be nice if I could simply enter a term on the command line and have my computer automatically open the top search results in new browser tabs.

Let's write a script to do this for the search results page of the Python Package Index at *https://pypi.org*. You could adapt a program like this to many other websites, although Google, DuckDuckGo, Amazon, and other large websites often employ measures that make scraping their search results pages difficult.

This is what the program should do:

- Get search keywords from the command line arguments
- Retrieve the search results page
- Open a browser tab for each result

This means your code needs to do the following:

- Read the command line arguments from `sys.argv`.
- Fetch the search results page with the `requests` module.
- Find the links to each search result.
- Call the `webbrowser.open()` function to open the web browser.
- Open a new file editor tab and save it as *searchpypi.py*.

Step 1: Get the Search Page

Before writing code, you first need to know the URL of the search results page. By looking at the browser's address bar after doing a search, you can see that the results page has a URL that looks like this: *https://pypi.org/search/?q=<SEARCH_TERM_HERE>*. The `requests` module can download this page; then, you can use Beautiful Soup to find the search result links in the HTML. Finally, you'll use the `webbrowser` module to open those links in browser tabs.

Make your code look like the following:

```
# searchpypi.py - Opens several search results on pypi.org

import requests, sys, webbrowser, bs4

print('Searching...')  # Display text while downloading the search results page.
res = requests.get('https://pypi.org/search/?q=' + ' '.join(sys.argv[1:]))
res.raise_for_status()

# TODO: Retrieve top search result links.

# TODO: Open a browser tab for each result.
```

The user will specify the search terms as command line arguments when launching the program, and the code stores these arguments as strings in a list in sys.argv.

Step 2: Find All Results

Now you need to use Beautiful Soup to extract the top search result links from your downloaded HTML. But how do you figure out the right selector for the job? For example, you can't just search for all <a> tags, because there are lots of links you don't care about in the HTML. Instead, you must inspect the search results page with the browser's Developer Tools to try to find a selector that will pick out only the links you want.

After doing a search for *pyautogui*, you can open the browser's Developer Tools and inspect some of the link elements on the page. They can look complicated, like pages of this: <a class="package-snippet" href="/project/pyautogui" >. But it doesn't matter that the element looks incredibly complicated. You just need to find the pattern that all the search result links have.

Make your code look like the following:

```
# searchpypi.py - Opens several search results on pypi.org
import requests, sys, webbrowser, bs4
--snip--
# Retrieve top search result links.
soup = bs4.BeautifulSoup(res.text, 'parser.html')
# Open a browser tab for each result.
link_elems = soup.select('.package-snippet')
```

If you look at the <a> elements, you'll see that the search result links all have class="package-snippet". Looking through the rest of the HTML source, it looks like the package-snippet class is used only for search result links. You don't have to know what the CSS class package-snippet is or what it does. You're just going to use it as a marker for the <a> element you're looking for.

You can create a BeautifulSoup object from the downloaded page's HTML text and then use the selector '.package-snippet' to find all <a> elements that are within an element that has the package-snippet CSS class. Note that if the PyPI website changes its layout, you may need to update this program with a new CSS selector string to pass to soup.select(). The rest of the program should remain up-to-date.

Step 3: Open Web Browsers for Each Result

Finally, you must tell the program to open web browser tabs for the results. Add the following to the end of your program:

```
# searchpypi.py - Opens several search results on pypi.org
import requests, sys, webbrowser, bs4
--snip--
# Open a browser tab for each result.
link_elems = soup.select('.package-snippet')
num_open = min(5, len(link_elems))
```

```
for i in range(num_open):
    url_to_open = 'https://pypi.org' + link_elems[i].get('href')
    print('Opening', url_to_open)
    webbrowser.open(url_to_open)
```

By default, the program opens the first five search results in new tabs using the webbrowser module. However, the user may have searched for something that turned up fewer than five results. The soup.select() call returns a list of all the elements that matched your '.package-snippet' selector, so the number of tabs you want to open is either 5 or the length of this list (whichever is smaller).

The built-in Python function min() returns the smallest of the integer or float arguments it is passed. (There is also a built-in max() function that returns the largest argument it is passed.) You can use min() to find out whether there are fewer than five links in the list and store the number of links to open in a variable named num_open. Then, you can run through a for loop by calling range(num_open).

On each iteration of the loop, the code uses webbrowser.open() to open a new tab in the web browser. Note that the href attribute's value in the returned <a> elements don't have the initial *https://pypi.org* part, so you have to concatenate that to the href attribute's string value.

Now you can instantly open the first five PyPI search results for, say, *boring stuff* by running searchpypi boring stuff on the command line! See Chapter 12 for how to easily run programs on your operating system.

Ideas for Similar Programs

The benefit of tabbed browsing is that you can easily open links in new tabs to peruse later. A program that automatically opens several links at once can be a nice shortcut to do the following:

- Open all the product pages after searching a shopping site such as Amazon.
- Open all the links to reviews for a single product.
- Open the result links to photos after performing a search on a photo site such as Flickr or Imgur.

Project 8: Download XKCD Comics

Blogs, web comics, and other regularly updating websites usually have a front page with the most recent post, as well as a Previous button on the page that takes you to the previous post. That post will also have a Previous button, and so on, creating a trail from the most recent page to the first post on the site. If you wanted a copy of the site's content to read when you're not online, you could manually navigate over every page and save each one. But this is pretty boring work, so let's write a program to do it instead.

XKCD, shown in Figure 13-5, is a popular geek webcomic with a website that fits this structure. The front page at *https://xkcd.com* has a Prev button

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

that guides the user back through prior comics. Downloading each comic by hand would take forever, but you can write a script to do this in a couple of minutes.

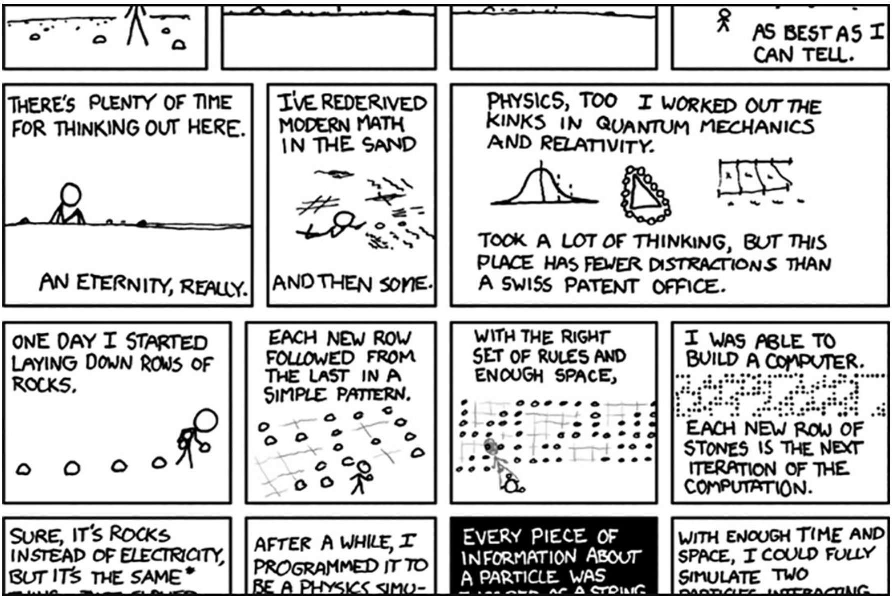

AS BEST AS I CAN TELL.
THERE'S PLENTY OF TIME FOR THINKING OUT HERE.
AN ETERNITY, REALLY.
I'VE REDERIVED MODERN MATH IN THE SAND
AND THEN SOME.
PHYSICS, TOO I WORKED OUT THE KINKS IN QUANTUM MECHANICS AND RELATIVITY.
TOOK A LOT OF THINKING, BUT THIS PLACE HAS FEWER DISTRACTIONS THAN A SWISS PATENT OFFICE.
ONE DAY I STARTED LAYING DOWN ROWS OF ROCKS.
EACH NEW ROW FOLLOWED FROM THE LAST IN A SIMPLE PATTERN.
WITH THE RIGHT SET OF RULES AND ENOUGH SPACE,
I WAS ABLE TO BUILD A COMPUTER.
EACH NEW ROW OF STONES IS THE NEXT ITERATION OF THE COMPUTATION.
SURE, IT'S ROCKS INSTEAD OF ELECTRICITY, BUT IT'S THE SAME*
AFTER A WHILE, I PROGRAMMED IT TO BE A PHYSICS SIMU-
EVERY PIECE OF INFORMATION ABOUT A PARTICLE WAS
WITH ENOUGH TIME AND SPACE, I COULD FULLY SIMULATE TWO

Figure 13-5: XKCD, "a webcomic of romance, sarcasm, math, and language"

Here's what your program should do:

- Load the XKCD home page.
- Save the comic image on that page.
- Follow the Previous Comic link.
- Repeat until it reaches the first comic or the max download limit.

This means your code will need to do the following:

- Download pages with the `requests` module.
- Find the URL of the comic image for a page using Beautiful Soup.
- Download and save the comic image to the hard drive with `iter_content()`.
- Find the URL of the Previous Comic link, and repeat.

Open a new file editor tab and save it as *downloadXkcdComics.py*.

Step 1: Design the Program

If you open the browser's Developer Tools and inspect the elements on the page, you should find the following to be true:

- The `href` attribute of an `<img>` element stores the URL of the comic's image file.

- The `<img>` element is inside a `<div id="comic">` element.
- The Prev button has a `rel` HTML attribute with the value `prev`.
- The oldest comic's Prev button links to the *https://xkcd.com/#* URL, indicating that there are no more previous pages.

To prevent the readers of this book from eating up too much of the XKCD website's bandwidth, let's limit the number of downloads we make to 10 by default. Make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD comics

import requests, os, bs4, time

url = 'https://xkcd.com'  # Starting URL
os.makedirs('xkcd', exist_ok=True)  # Store comics in ./xkcd
num_downloads = 0
MAX_DOWNLOADS = 10
while not url.endswith('#') and num_downloads < MAX_DOWNLOADS:
    # TODO: Download the page.

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

The program creates a `url` variable that starts with the value `'https://xkcd.com'` and repeatedly updates it (in a `while` loop) with the URL of the current page's Prev link. At every step in the loop, you'll download the comic at `url`. The loop stops when `url` ends with `'#'` or you have downloaded `MAX_DOWNLOADS` comics.

You'll download the image files to a folder in the current working directory named *xkcd*. The call `os.makedirs()` ensures that this folder exists, and the `exist_ok=True` keyword argument prevents the function from throwing an exception if this folder has already been created.

Step 2: Download the Web Page

Let's implement the code for downloading the page. Make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD comics

import requests, os, bs4, time

url = 'https://xkcd.com'  # Starting URL
os.makedirs('xkcd', exist_ok=True)  # Store comics in ./xkcd
num_downloads = 0
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
MAX_DOWNLOADS = 10
while not url.endswith('#') and num_downloads < MAX_DOWNLOADS:
    # Download the page.
    print(f'Downloading page {url}...')
    res = requests.get(url)
    res.raise_for_status()

    soup = bs4.BeautifulSoup(res.text, 'html.parser')

    # TODO: Find the URL of the comic image.

    # TODO: Download the image.

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

First, print `url` so that the user knows which URL the program is about to download; then, use the `requests` module's `requests.get()` function to download it. As always, you should immediately call the `Response` object's `raise_for_status()` method to throw an exception and end the program if something went wrong with the download. Otherwise, create a `BeautifulSoup` object from the text of the downloaded page.

Step 3: Find and Download the Comic Image

To download the comic on each page, make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD comics

import requests, os, bs4, time

--snip--

    # Find the URL of the comic image.
    comic_elem = soup.select('#comic img')
    if comic_elem == []:
        print('Could not find comic image.')
    else:
        comic_URL = 'https:' + comic_elem[0].get('src')
        # Download the image.
        print(f'Downloading image {comic_URL}...')
        res = requests.get(comic_URL)
        res.raise_for_status()

    # TODO: Save the image to ./xkcd.

    # TODO: Get the Prev button's url.

print('Done.')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Because you inspected the XKCD home page with your Developer Tools, you know that the `<img>` element for the comic image is inside another element with the `id` attribute set to `comic`, so the selector `'#comic img'` will get you the correct `<img>` element from the `BeautifulSoup` object.

A few XKCD pages have special content that isn't a simple image file. That's fine; you'll just skip those. If your selector doesn't find any elements, `soup.select('#comic img')` will return a `ResultSet` object of a blank list. When that happens, the program can just print an error message and move on without downloading the image.

Otherwise, the selector will return a list containing one `<img>` element. You can get the `src` attribute from this `<img>` element and pass it to `requests.get()` to download the comic's image file.

Step 4: Save the Image and Find the Previous Comic

At this point, the comic's image file is stored in the `res` variable. You need to write this image data to a file on the hard drive. Make your code look like the following:

```
# downloadXkcdComics.py - Downloads XKCD comics

import requests, os, bs4, time

--snip--

    # Save the image to ./xkcd.
        image_file = open(os.path.join('xkcd', os.path.basename(comic_URL)), 'wb')
            for chunk in res.iter_content(100000):
               image_file.write(chunk)
            image_file.close()

    # Get the Prev button's URL.
      prev_link = soup.select('a[rel="prev"]')[0]
      url = 'https://xkcd.com' + prev_link.get('href')
      num_downloads += 1
      time.sleep(1)  # Pause so we don't hammer the web server.

print('Done.')
```

You'll also need a filename for the local image file to pass to `open()`. The `comic_URL` will have a value like `'https://imgs.xkcd.com/comics/heartbleed_explanation.png'`, which you might have noticed looks a lot like a filepath. In fact, you can call `os.path.basename()` with `comic_URL` to return just the last part of the URL, `'heartbleed_explanation.png'`, and use this as the filename when saving the image to your hard drive. Join this name with the name of your `xkcd` folder using `os.path.join()` so that your program uses backslashes (\) on Windows and forward slashes (/) on macOS and Linux. Now that you finally have the filename, you can call `open()` to open a new file in `'wb'` mode.

Remember from earlier in this chapter that, to save files you've downloaded using `requests`, you need to loop over the return value of the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

iter_content() method. The code in the for loop writes chunks of the image data to the file. Then, the code closes the file, saving the image to your hard drive.

Afterward, the selector 'a[rel="prev"]' identifies the <a> element with the rel attribute set to prev. You can use this <a> element's href attribute to get the previous comic's URL, which gets stored in url.

The last part of the loop's code increments num_downloads by 1 so that it doesn't download all of the comics by default. It also introduces a one-second pause with time.sleep(1) to prevent the script from "hammering" the site (that is, impolitely downloading comics as fast as possible, which may cause performance issues for other website visitors). Then, the while loop begins the entire download process again.

The output of this program will look like this:

```
Downloading page https://xkcd.com...
Downloading image https://imgs.xkcd.com/comics/phone_alarm.png...
Downloading page https://xkcd.com/1358/...
Downloading image https://imgs.xkcd.com/comics/nro.png...
Downloading page https://xkcd.com/1357/...
Downloading image https://imgs.xkcd.com/comics/free_speech.png...
Downloading page https://xkcd.com/1356/...
Downloading image https://imgs.xkcd.com/comics/orbital_mechanics.png...
Downloading page https://xkcd.com/1355/...
Downloading image https://imgs.xkcd.com/comics/airplane_message.png...
Downloading page https://xkcd.com/1354/...
Downloading image https://imgs.xkcd.com/comics/heartbleed_explanation.png...
--snip--
```

This project is a good example of a program that can automatically follow links to scrape large amounts of data from the web. You can learn about Beautiful Soup's other features from its documentation at *https://www.crummy.com/software/BeautifulSoup/bs4/doc/*.

Ideas for Similar Programs

Many web crawling programs involve downloading pages and following links. Similar programs could do the following:

- Back up an entire site by following all of its links.
- Copy all the messages on a web forum.
- Duplicate the catalog of items for sale on an online store.

The requests and bs4 modules are great as long as you can figure out the URL you need to pass to requests.get(). However, this URL isn't always so easy to find. Or perhaps the website you want your program to navigate requires you to log in first. Selenium will give your programs the power to perform such sophisticated tasks.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Controlling the Browser with Selenium

Selenium lets Python directly control the browser by programmatically clicking links and filling in forms, just as a human user would. Using Selenium, you can interact with web pages in a much more advanced way than with `requests` and Beautiful Soup; but because it launches a web browser, it's a bit slower and hard to run in the background if, say, you just need to download some files from the web.

Still, if you need to interact with a web page in a way that, for instance, depends on the JavaScript code that updates the page, you'll need to use Selenium instead of `requests`. That's because major e-commerce websites such as Amazon almost certainly have software systems to recognize traffic that they suspect is a script harvesting their info or signing up for multiple free accounts. These sites may refuse to serve pages to you after a while, breaking any scripts you've made. Selenium is much more likely than `requests` to function on these sites long term.

A major "tell" to websites that you're using a script is the *user-agent* string, which identifies the web browser and is included in all HTTP requests. For example, the user-agent string for the `requests` module is something like `'python-requests/X.XX.X'`. You can visit a site such as *https://www.whatsmyua.info* to see your user-agent string. Using Selenium, you're much more likely to pass for human, because not only is Selenium's user agent the same as a regular browser (for instance, `' Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:108.0) Gecko/20100101 Firefox/108.0'`), but it has the same traffic patterns: a Selenium-controlled browser will download images, advertisements, cookies, and privacy-invading trackers just like a regular browser. However, websites can still find ways to detect Selenium, and major ticketing and e-commerce websites often block it to prevent the web scraping of their pages.

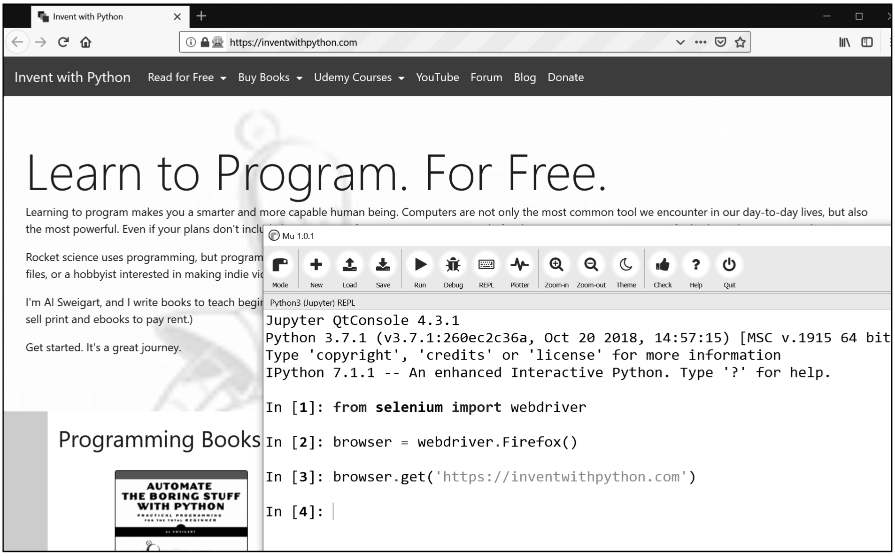
Starting a Selenium-Controlled Browser

The following examples will show you how to control Firefox's web browser. If you don't already have Firefox, you can download it for free from *https://getfirefox.com.*

Importing Selenium's modules is slightly tricky. Instead of `import selenium`, you must run **`from selenium import webdriver`**. (The exact reason why Selenium is set up this way is beyond the scope of this book.) After that, you can launch the Firefox browser with Selenium. Enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> browser = webdriver.Firefox()
>>> type(browser)
<class 'selenium.webdriver.firefox.webdriver.WebDriver'>
>>> browser.get('https://inventwithpython.com')
```

You'll notice that when `webdriver.Firefox()` is called, the Firefox web browser starts up. Calling `type()` on the value `webdriver.Firefox()` reveals it's of the `WebDriver` data type. And calling `browser.get('https://inventwithpython.com')` directs the browser to *https://inventwithpython.com*. Your browser should look something like Figure 13-6.



Learn to Program. For Free.

Learning to program makes you a smarter and more capable human being. Computers are not only the most common tool we encounter in our day-to-day lives, but also the most powerful. Even if your plans don't inclu

Rocket science uses programming, but program files, or a hobbyist interested in making indie vi

I'm Al Sweigart, and I write books to teach begi sell print and ebooks to pay rent.)

Get started. It's a great journey.

Programming Books

```
Jupyter QtConsole 4.3.1
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit
Type 'copyright', 'credits' or 'license' for more information
IPython 7.1.1 -- An enhanced Interactive Python. Type '?' for help.

In [1]: from selenium import webdriver

In [2]: browser = webdriver.Firefox()

In [3]: browser.get('https://inventwithpython.com')

In [4]:
```

Figure 13-6: After we call `webdriver.Firefox()` and `get()` in Mu, the Firefox browser appears.

If you encounter the error message "geckodriver executable needs to be in PATH," you need to manually download the web driver for Firefox before you can use Selenium to control it. You can also control browsers other than Firefox if you install the web driver for them, and instead of manually downloading browser web drivers, you can use the webdriver-manager package from *https://pypi.org/project/webdriver-manager/*.

Clicking Browser Buttons

Selenium can simulate clicks on various browser buttons through the following methods:

`browser.back()` Clicks the Back button

`browser.forward()` Clicks the Forward button

`browser.refresh()` Clicks the Refresh/Reload button

`Browser.quit()` Clicks the Close Window button

Finding Elements on the Page

A `WebDriver` object has the `find_element()` and `find_elements()` methods for finding elements on a web page. The `find_element()` method returns a single `WebElement` object, representing the first element on the page that matches your query. The `find_elements()` method returns a list of `WebElement` objects for *every* matching element on the page.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You can find elements through their class name, CSS selector, ID, or another means. First, run **`from selenium.webdriver.common.by import By`** to get the `By` object. The `By` object has several constants you can pass to the `find_element()` and `find_elements()` methods. Table 13-3 lists these constants.

Table 13-3: Selenium's `By` Constants for Finding Elements

Constant name	`WebElement` object/list returned
`By.CLASS_NAME`	Elements that use the CSS class *name*
`By.CSS_SELECTOR`	Elements that match the CSS *`selector`*
`By.ID`	Elements with a matching *`id`* attribute value
`By.LINK_TEXT`	<a> elements that completely match the *`text`* provided
`By.PARTIAL_LINK_TEXT`	<a> elements that contain the *`text`* provided
`By.NAME`	Elements with a matching *name* attribute value
`By.TAG_NAME`	Elements with a matching tag *name* (case-insensitive; an <a> element is matched by `'a'` and `'A'`)

If no elements exist on the page that match what the method is looking for, Selenium raises a `NoSuchElement` exception. If you do not want this exception to crash your program, add `try` and `except` statements to your code.

Once you have the `WebElement` object, you can learn more about it by reading the attributes or calling the methods in Table 13-4.

Table 13-4: `WebElement` Attributes and Methods

Attribute or method	Description
`tag_name`	The tag name, such as `'a'` for an <a> element.
`get_attribute(`*`name`*`)`	The value for the element's `name` attribute, like `href` in an <a> element.
`get_property(`*`name`*`)`	The value for the element's property, which does not appear in the HTML code. Some examples of HTML properties are `innerHTML` and `innerText`.
`text`	The text within the element, such as `'hello'` in the following: <span>hello </span>
`clear()`	For text field or text area elements, clears the text entered into it.
`is_displayed()`	Returns `True` if the element is visible; otherwise, returns `False`.
`is_enabled()`	For input elements, returns `True` if the element is enabled; otherwise, returns `False`.
`is_selected()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise, returns `False`.
`location`	A dictionary with keys `'x'` and `'y'` for the position of the element in the page.
`size`	A dictionary with keys `'width'` and `'height'` for the size of the element in the page.

For example, open a new file editor tab and enter the following program:

```
from selenium import webdriver
from selenium.webdriver.common.by import By
browser = webdriver.Firefox()
browser.get('https://autbor.com/example3.html')
elems = browser.find_elements(By.CSS_SELECTOR, 'p')
print(elems[0].text)
print(elems[0].get_property('innerHTML'))
```

Here, we open Firefox and direct it to a URL. On this page, we get a list of the `<p>` elements, look at the first one at index `0`, and then get the string of the text inside that `<p>` element. Next, we get the string of its `innerHTML` property. This program outputs the following:

```
This <p> tag puts content into a single paragraph.
This &lt;p&gt; tag puts <b>content</b> into a <i>single</i> paragraph.
```

The element's `text` attribute shows the text as we'd see it in the web browser: "This `<p>` tag puts content into a single paragraph." We can also examine the element's `innerHTML` property by calling the `get_property()` method, which is the HTML source code that includes tags and HTML entities. (The `&lt;` and `&gt;` are HTML escape characters that represent the less than [<] and greater than [>] characters.)

Note that the `text` attribute is just a shortcut for calling `get_property('innerText')`. The names *innerHTML* and *innerText* are standard names of properties for HTML elements. In short, these element properties are accessed by JavaScript code and web drivers, while element attributes are part of the HTML source code, like the `href` in `<a href="https://inventwithpython.com">`.

Clicking Elements on the Page

The `WebElement` objects returned from the `find_element()` and `find_elements()` methods have a `click()` method that simulates a mouse click on that element. This method can be used to follow a link, make a selection on a radio button, click a Submit button, or trigger whatever else might happen when a mouse clicks the element. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.by import By
>>> browser = webdriver.Firefox()
>>> browser.get('https://autbor.com/example3.html')
>>> link_elem = browser.find_element(By.LINK_TEXT, 'This text is a link')
>>> type(link_elem)
<class 'selenium.webdriver.remote.webelement.WebElement'>
>>> link_elem.click()  # Follows the "This text is a link" link
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

This code opens Firefox to *https://autbor.com/example3.html*, gets the `WebElement` object for the `<a>` element with the text *This is a link*, and then simulates clicking that `<a>` element as if you'd clicked the link yourself; the browser then follows that link.

Filling Out and Submitting Forms

Sending keystrokes to text fields on a web page is a matter of finding the `<input>` or `<textarea>` element for that text field and then calling the `send_keys()` method. For example, enter the following into the interactive shell:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.by import By
>>> browser = webdriver.Firefox()
>>> browser.get('https://autbor.com/example3.html')
>>> user_elem = browser.find_element(By.ID, 'login_user')
>>> user_elem.send_keys('your_real_username_here')
>>> password_elem = browser.find_element(By.ID, 'login_pass')
>>> password_elem.send_keys('your_real_password_here')
>>> password_elem.submit()
```

As long as the login page hasn't changed the `id` of the username and password `<input>` elements, the previous code will fill in those text fields with the provided text. (You can always use the browser's inspector to verify the `id`.) Calling the `submit()` method on any element will have the same result as clicking the Submit button for the form that element is in. (You could have just as easily called `user_elem.submit()`, and the code would have done the same thing.)

WARNING *Avoid putting your passwords in source code whenever possible. It's easy to accidentally leak your passwords to others when they are left unencrypted on your hard drive.*

Sending Special Keys

Selenium has a module, `selenium.webdriver.common.keys`, to represent keyboard keys, which it stores in attributes. Because the module has such a long name, it's much easier to run `from selenium.webdriver.common.keys import Keys` at the top of your program; if you do, you can simply write `Keys` anywhere you'd normally have to write `selenium.webdriver.common.keys`.

You can pass `send_keys()` any of the following constants:

```
Keys.ENTER            Keys.PAGE_UP          Keys.DOWN
Keys.RETURN           Keys.ESCAPE           Keys.LEF
Keys.HOME             Keys.BACK_SPACE       Keys.RIGHT
Keys.END              Keys.DELETE           Keys.TAB
Keys.PAGE_DOWN        Keys.UP               Keys.F1 to Keys.F12
```

You can also pass the method a string, such as `'hello'` or `'?'`.

For example, if the cursor isn't currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

page, respectively. Enter the following into the interactive shell, and notice how the send_keys() calls scroll the page:

```
>>> from selenium import webdriver
>>> from selenium.webdriver.common.by import By
>>> from selenium.webdriver.common.keys import Keys
>>> browser = webdriver.Firefox()
>>> browser.get('https://nostarch.com')
>>> html_elem = browser.find_element(By.TAG_NAME, 'html')
>>> html_elem.send_keys(Keys.END)  # Scrolls to bottom
>>> html_elem.send_keys(Keys.HOME)  # Scrolls to top
```

The <html> tag is the base tag in HTML files: the full content of the HTML file is enclosed within the <html> and </html> tags. Calling browser.find_element(By.TAG_NAME, 'html') is a good place to send keys to the general web page via the main <html> tag. This would be useful if, for example, new content is loaded once you've scrolled to the bottom of the page.

Selenium can do much more than the functions described here. It can modify your browser's cookies, take screenshots of web pages, and run custom JavaScript. To learn more about these features, you can visit the Selenium documentation at *https://selenium-python.readthedocs.io*. You can also find Python conference talks on Selenium by searching the website *https://pyvideo.org*.

Controlling the Browser with Playwright

Playwright is a browser-controlling library similar to Selenium, but it's newer. While it might not currently have the wide audience that Selenium has, it does offer some features that merit learning. Chief among these new features is the ability to run in *headless mode*, meaning you can simulate a browser without actually having the browser window open on your screen. This makes it useful for running automated tests or web scraping jobs in the background. Playwright's full documentation is at *https://playwright.dev/python/docs/intro*.

Also, installing web drivers for individual browsers is easier to do with Playwright compared to Selenium: just run **python -m playwright install** on Windows and **python3 -m playwright install** on macOS and Linux from a terminal window to install the web drivers for Firefox, Chrome, and Safari. Because Playwright is otherwise similar to Selenium, I won't cover the general web scraping and CSS selector information in this section.

Starting a Playwright-Controlled Browser

Once Playwright is installed, you can test it with the following program:

```
from playwright.sync_api import sync_playwright
with sync_playwright() as playwright:
    browser = playwright.firefox.launch()
    page = browser.new_page()
    page.goto('https://autbor.com/example3.html')
    print(page.title())
    browser.close()
```

When run, this program pauses while it loads the Firefox browser and the *https://autbor.com/example3.html* website, and then prints its title, "Example Website." You can also use `playwright.chromium.launch()` or `playwright.webkit.launch()` to use the Chrome and Safari browsers, respectively.

Playwright automatically calls the `start()` and `stop()` methods when the execution enters and exits the `with` statement's block. Playwright has a synchronous mode, where its functions don't return until the operation is complete. This way, you don't accidentally tell the browser to find an element before the page has finished loading. Playwright's asynchronous features are beyond the scope of this book.

You may have noticed that no browser window appeared at all, because, by default, Playwright runs in headless mode. This, along with how Playwright puts its code inside a `with` statement, can make debugging tricky. To run Playwright one step at a time, enter the following into the interactive shell:

```
>>> from playwright.sync_api import sync_playwright
>>> playwright = sync_playwright().start()
>>> browser = playwright.firefox.launch(headless=False, slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/example3.html')
<Response url='https://autbor.com/example3.html' request=<Request
 url='https://autbor.com/example3.html' method='GET'>>
>>> browser.close()
>>> playwright.stop()
```

The `headless=False` and `slow_mo=50` keyword arguments to `playwright.firefox.launch()` make the browser window appear on your screen and add a 50 ms delay to its operations so that it's easier for you to see what is happening. You don't have to worry about adding pauses to give web pages time to load: Playwright is much better than Selenium about not moving on to new operations before the previous one has finished.

The `Page` object returned by the `new_page()` `Browser` method represents a new tab in a new browser window. You can have multiple browser windows open at the same time when using Playwright.

Clicking Browser Buttons

Playwright can simulate clicking the browser buttons by calling the following `Page` methods on the `Page` object returned by `browser.new_page()`:

`page.go_back()` Clicks the Back button

`page.go_forward()` Clicks the Forward button

`page.reload()` Clicks the Refresh/Reload button

`page.close()` Clicks the Close Window button

Finding Elements on the Page

Playwright has `Page` object methods colloquially called *locators* that return `Locator` objects, which represent possible HTML elements on a web page. I say *possible* because, while Selenium immediately raises an error if it can't find the element you ask for, Playwright understands that the page might dynamically create the element later. This is useful but has a slightly unfortunate side effect: if the element you specified doesn't exist, Playwright pauses for 30 seconds while it waits for the element to appear.

But this 30-second pause is tedious if you've simply made a typo. To immediately check whether an element exists and is visible on the page, call the `is_visible()` method on the `Locator` object returned by the locator. You can also call `page.query_selector('`*`selector`*`')` where *`selector`* is a string of the element's CSS or XPath selector. The `page.query_selector()` method immediately returns, and if it returns `None`, the element doesn't currently exist on the page. A `Locator` object may match one or more HTML elements on the web page. Table 13-5 contains Playwright's locators.

Table 13-5: Playwright's Locators for Finding Elements

Locator	`Locator` object returned
`page.get_by_role(`*`role, name`*`=label)`	Elements by their role and optionally their *`label`*
`page.get_by_text(`*`text`*`)`	Elements that contain *`text`* as part of their inner text
`page.get_by_label(`*`label`*`)`	Elements with matching `<`*`label`*`>` text as *`label`*
`page.get_by_placeholder(`*`text`*`)`	`<input>` and `<textarea>` elements with matching `placeholder` attribute values as the *`text`* provided
`page.get_by_alt_text(`*`text`*`)`	`<img>` elements with matching *`alt`* attribute values as the *`text`* provided
`page.locator(`*`selector`*`)`	Elements with a matching CSS or XPath *`selector`*

The `get_by_role()` method makes use of *Accessible Rich Internet Applications (ARIA)* roles, a set of standards that enable software to identify web page content to adapt it for users with vision or other disabilities. For example, the "heading" role applies to the `<h1>` through `<h6>` tags, with the text in between `<h1>` and `</h1>` being the text you can identify with the `get_by_role()` method's `name` keyword parameter. (There is much more to ARIA roles than this, but the topic is beyond the scope of this book.)

You can use the text between the starting and ending tags to locate elements. Calling `page.get_by_text('is a link')` would locate the `<a>` element in `<a href="https://inventwithpython.com">This text is a link</a>`. A partial, case-insensitive text match is generally good enough to locate the element.

The `page.get_by_label()` method locates elements using the text between `<label>` and `</label>` tags. For example, `page.get_by_label('Agree')` would locate the `<input>` checkbox element in `<label>Agree to disagree: <input type="checkbox" /></label>`.

The `<input>` and `<textarea>` tags can have a `placeholder` attribute to show placeholder text until the user enters real text. For example, `page.get_by_placeholder('admin')` would locate the `<input>` element for `<input id="login_user" placeholder="admin" />`.

Images on web pages can have alt text in their `alt` attribute to describe the image contents to sight-impaired users. Some browsers show the alt text as a tool tip if you hover the mouse cursor over the image. The `page.get_by_alt_text('Zophie')` call would return the `<img>` element in `<img src="wow_such_zophie_thumb.webp" alt="Close-up of my cat Zophie." />`.

If you just need to obtain a `Locator` object via a CSS selector, call the `locator()` locator and pass it the selector string. This is similar to Selenium's `find_elements()` method with the `By.CSS_SELECTOR` constant.

Table 13-6: `Locator` Methods

Method	Description
`get_attribute(`*name*`)`	Returns the value for the element's *name* attribute, such as `'https://nostarch.com'` for the `href` attribute in an `<a href="https://nostarch.com">` element
`count()`	Returns an integer of the number of matching elements in this `Locator` object
`nth(`*index*`)`	Returns a `Locator` object of the matching element given by the index. For example, `nth(3)` returns the fourth matching element since index `0` is the first matching element.
`first`	The `Locator` object of the first matching element. This is the same as `nth(0)`.
`last`	The `Locator` object of the last matching element. If there are, say, five match elements, this is the same as `nth(4)`.
`all()`	Returns a list of `Locator` objects for each individual matching element
`inner_text()`	Returns the text within the element, such as `'hello'` in `<b>hello</b>`
`inner_html()`	Returns the HTML source within the element, such as `'<b>hello</b>'` in `<b>hello</b>`
`click()`	Simulates a click on the element, which is useful for link, checkbox, and button elements
`is_visible()`	Returns `True` if the element is visible; otherwise, returns `False`
`is_enabled()`	For input elements, returns `True` if the element is enabled; otherwise, returns `False`
`is_checked()`	For checkbox or radio button elements, returns `True` if the element is selected; otherwise, returns `False`
`bounding_box()`	Returns a dictionary with keys `'x'` and `'y'` for the position of the element's top-left corner in the page, along with keys `'width'` and `'height'` for the element's size

Since `Locator` objects can represent multiple elements, you can obtain a `Locator` object for an individual element with the `nth()` method, passing the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

zero-based index. For example, open a new file editor tab and enter the following program:

```
from playwright.sync_api import sync_playwright
with sync_playwright() as playwright:
    browser = playwright.firefox.launch(headless=False, slow_mo=50)
    page = browser.new_page()
    page.goto('https://autbor.com/example3.html')
    elems = page.locator('p')
    print(elems.nth(0).inner_text())
    print(elems.nth(0).inner_html())
```

Like the Selenium example, this program outputs the following:

```
This <p> tag puts content into a single paragraph.
This &lt;p&gt; tag puts <b>content</b> into a <i>single</i> paragraph.
```

The `page.locator('p')` code returns a `Locator` object that matches all `<p>` elements in the web page, and the `nth(0)` method call returns a `Locator` object for just the first `<p>` element. The `Locator` objects also have a `count()` method for returning the number of matching elements in the locator (similar to the `len()` function for Python lists). There are also `first` and `last` attributes that contain a locator that matches the first or last element. If you want a list of `Locator` objects for each individual matching element, call the `all()` method.

Once you have `Locator` objects for elements, you can perform mouse clicks and key presses on them, as described in the next few sections.

Clicking Elements on the Page

The `Page` object has `click()`, `check()`, `uncheck()`, and `set_checked()` methods for simulating clicks on link, button, and checkbox elements. You can call these methods and pass the string of a CSS or XPath selector of the element, or you can use Playwright's `Locator` functions in Table 13-6. Enter the following into the interactive shell:

```
>>> from playwright.sync_api import sync_playwright
>>> playwright = sync_playwright().start()
>>> browser = playwright.firefox.launch(headless=False, slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/example3.html')
<Response url='https://autbor.com/example3.html' request=<Request
 url='https://autbor.com/example3.html' method='GET'>>
>>> page.click('input[type=checkbox]')  # Checks the checkbox.
>>> page.click('input[type=checkbox]')  # Unchecks the checkbox.
>>> page.click('a')  # Clicks the link.
>>> page.go_back()
>>> checkbox_elem = page.get_by_role('checkbox')  # Calls a Locator method.
>>> checkbox_elem.check()  # Checks the checkbox.
>>> checkbox_elem.uncheck()  # Unchecks the checkbox.
>>> checkbox_elem.set_checked(True)  # Checks the checkbox.
>>> checkbox_elem.set_checked(False)  # Unchecks the checkbox.
>>> page.get_by_text('is a link').click()  # Uses a Locator method.
>>> browser.close()
>>> playwright.stop()
```

The check() and uncheck() methods are more reliable than click() for checkboxes. The click() method toggles the checkbox to the opposite state, while check() and uncheck() leave them checked or unchecked no matter what state they were in before. Similarly, the set_checked() method allows you to pass True to check the checkbox or False to uncheck it.

Filling Out and Submitting Forms

Locator objects have a fill() method that takes a string and fills in the <input> or <textarea> element with the text. This is useful for filling out online forms, such as the login form in our *example3.html* web page:

```
>>> from playwright.sync_api import sync_playwright
>>> playwright = sync_playwright().start()
>>> browser = playwright.firefox.launch(headless=False, slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/example3.html')
<Response url='https://autbor.com/example3.html' request=<Request
url='https://autbor.com/example3.html' method='GET'>>
>>> page.locator('#login_user').fill('your_real_username_here')
>>> page.locator('#login_pass').fill('your_real_password_here')
>>> page.locator('input[type=submit]').click()
>>> browser.close()
>>> playwright.stop()
```

There's also a clear() method, which erases all of the text currently in the element. Unlike in Selenium, there's no submit() method in Playwright, and you'll have to call click() on its Locator object matching the Submit button's element.

Sending Special Keys

You can also simulate keyboard key presses on elements in the web page with the press() method for Locator objects. For example, if the cursor isn't currently in a text field, pressing the HOME and END keys will scroll the browser to the top and bottom of the page, respectively. Enter the following into the interactive shell, and notice how the press() calls scroll the page:

```
>>> from playwright.sync_api import sync_playwright
>>> playwright = sync_playwright().start()
>>> browser = playwright.firefox.launch(headless=False, slow_mo=50)
>>> page = browser.new_page()
>>> page.goto('https://autbor.com/example3.html')
<Response url='https://autbor.com/example3.html' request=<Request
url='https://autbor.com/example3.html' method='GET'>>
>>> page.locator('html').press('End')  # Scrolls to bottom
>>> page.locator('html').press('Home')  # Scrolls to top
>>> browser.close()
>>> playwright.stop()
```

The strings you pass to `press()` can include single character strings (such as `'a'` or `'?'`); the modification keys `'Shift'`, `'Control'`, `'Alt'`, or `'Meta'` (as in `'Control+A'`, for CTRL-A); and any of the following:

```
'Backquote'          'Escape'             'ArrowDown'
'Minus'              'End'                'ArrowRight'
'Equal'              'Enter'              'ArrowUp'
'Backslash'          'Home'               'F1' to 'F12'
'Backspace'          'Insert'             'Digit0' to 'Digit9'
'Tab'                'PageUp'             'KeyA' to 'KeyZ'
'Delete'             'PageDown'
```

Playwright can do much more beyond the functions described here. To learn more about these features, you can visit the Playwright documentation at *https://playwright.dev*. You can also find Python conference talks on Playwright by searching *https://pyvideo.org*.

Summary

Most boring tasks aren't limited to the files on your computer. Being able to programmatically download web pages will extend your programs to the internet. The `requests` module makes downloading straightforward, and with some basic knowledge of HTML concepts and selectors, you can utilize the `BeautifulSoup` module to parse the pages you download.

But to fully automate any web-based task, you need direct control of your web browser through the Selenium and Playwright packages. These packages will allow you to log in to websites and fill out forms automatically. Because a web browser is the most common way to send and receive information over the internet, this is a great ability to have in your programmer toolkit.

Practice Questions

1. Briefly describe the differences between the `webbrowser`, `requests`, and `bs4` modules.
2. What type of object is returned by `requests.get()`? How can you access the downloaded content as a string value?
3. What `requests` method checks that the download worked?
4. How can you get the HTTP status code of a `requests` response?
5. How do you save a `requests` response to a file?
6. What two formats do most online APIs return their responses in?
7. What is the keyboard shortcut for opening a browser's Developer Tools?
8. How can you view (in the Developer Tools) the HTML of a specific element on a web page?
9. What CSS selector string would find the element with an `id` attribute of `main`?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

10. What CSS selector string would find the elements with an `id` attribute of `highlight`?
11. Say you have a Beautiful Soup `Tag` object stored in the variable `spam` for the element `<div>Hello, world!</div>`. How could you get a string `'Hello, world!'` from the `Tag` object?
12. How would you store all the attributes of a Beautiful Soup `Tag` object in a variable named `link_elem`?
13. Running `import selenium` doesn't work. How do you properly import Selenium?
14. What's the difference between the `find_element()` and `find_elements()` methods in Selenium?
15. What methods do Selenium's `WebElement` objects have for simulating mouse clicks and keyboard keys?
16. In Playwright, what locator method call simulates pressing CTRL-A to select all the text on the page?
17. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Selenium?
18. How can you simulate clicking a browser's Forward, Back, and Refresh buttons with Playwright?

Practice Programs

For practice, write programs to do the following tasks.

Image Site Downloader

Write a program that goes to a photo-sharing site like Flickr or Imgur, searches for a category of photo, and then downloads all the resulting images. You could write a program that works with any photo site that has a search feature.

2048

The game 2048 is a simple game in which you combine tiles by sliding them up, down, left, or right with the arrow keys. You can actually get a fairly high score by sliding tiles in random directions. Write a program that will open the game at *https://play2048.co* and keep sending up, right, down, and left keystrokes to automatically play the game.

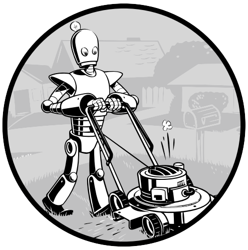
Link Verification

Write a program that, given the URL of a web page, will find every `<a>` link on the page and test whether the linked URL results in a "404 Not Found" status code. The program should print out any broken links.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

14

EXCEL SPREADSHEETS



Although we don't often think of spreadsheets as programming tools, almost everyone uses them to organize information into two-dimensional data structures, perform calculations with formulas, and produce output as charts. In the next two chapters, we'll integrate Python into two popular spreadsheet applications: Microsoft Excel and Google Sheets.

Excel is a popular and powerful spreadsheet application for Windows. The `openpyxl` module allows your Python programs to read and modify Excel spreadsheet files. For example, you might have the boring task of copying certain data from one spreadsheet and pasting it into another one. Or you might have to go through thousands of rows and pick out just a handful of them to make small edits based on some criteria. Or you might have to look through hundreds of spreadsheets of department budgets, searching for any that are in the red. These are exactly the sorts of boring, mindless spreadsheet tasks that Python can do for you.

Although Excel is proprietary software from Microsoft, LibreOffice is a free alternative that runs on Windows, macOS, and Linux. LibreOffice Calc uses Excel's *.xlsx* file format for spreadsheets, which means the `openpyxl` module can work on spreadsheets from this application as well. You can download it from *https://www.libreoffice.org*. Even if you already have Excel installed on your computer, you may find this program easier to use. The screenshots in this chapter, however, are all from the cloud-based Office 365 Excel.

The `openpyxl` module operates on Excel files, and not the desktop Excel application or cloud-based Excel web app. If you're using the cloud-based Office 365, you must click **File** ▸ **Save As** ▸ **Download a Copy** to download a spreadsheet, run your Python script to edit the spreadsheet file, and then re-upload the spreadsheet to Office 365 to see the changes. If you have the desktop Excel application, you must close the spreadsheet, run your Python script to edit the spreadsheet file, and then reopen it in Excel to see the changes.

Python does not come with `openpyxl`, so you'll have to install it. Appendix A has information on how to install third-party packages with Python's pip tool. You can find the full `openpyxl` documentation at *https://openpyxl.readthedocs.io/en/stable/*.

You'll use several example spreadsheet files in this chapter. You can download them from the book's online materials at *https://nostarch.com/automatestuff3*.

Reading Excel Files

First, let's go over some basic definitions. An Excel spreadsheet document is called a *workbook*. A single workbook is saved in a file with the *.xlsx* extension.

Each workbook can contain multiple *sheets* (also called *worksheets*). The sheet the user is currently viewing (or last viewed before closing Excel) is called the *active sheet*. Each sheet has *columns* (addressed by letters starting at *A*) and *rows* (addressed by numbers starting at 1). A box at a particular column and row is called a *cell*. Each cell can contain a number or text value. The grid of cells and their data makes up a sheet.

The examples in this chapter will use a spreadsheet named *example3.xlsx* stored in the current working directory. You can either create the spreadsheet yourself or download it from this book's online resources. Figure 14-1 shows the tabs for the three sheets named *Sheet1*, *Sheet2*, and *Sheet3*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

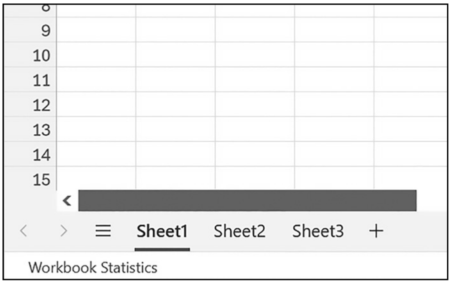


Figure 14-1: The tabs for a workbook's sheets are in the lower-left corner of Excel.

Sheet1 in the example file should look like Table 14-1. (If you didn't download *example3.xlsx*, you should enter this data into the sheet yourself.)

Table 14-1: The *example3.xlsx* Spreadsheet

	A	B	C
1	4/5/2035 1:34:02 PM	Apples	73
2	4/5/2035 3:41:23 AM	Cherries	85
3	4/6/2035 12:46:51 PM	Pears	14
4	4/8/2035 8:59:43 AM	Oranges	52
5	4/10/2035 2:07:00 AM	Apples	152
6	4/10/2035 6:10:37 PM	Bananas	23
7	4/10/2035 2:40:46 AM	Strawberries	98

Now that we have our example spreadsheet, let's see how we can manipulate it with the `openpyxl` module.

Opening a Workbook

Once you've imported the `openpyxl` module, you'll be able to open *.xlsx* files with the `openpyxl.load_workbook()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> type(wb)
<class 'openpyxl.workbook.workbook.Workbook'>
```

The `openpyxl.load_workbook()` function takes in the filename and returns a value of the `Workbook` data type. This `Workbook` object represents the Excel file, a bit like how a `File` object represents an opened text file.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Getting Sheets from the Workbook

You can get a list of all the sheet names in the workbook by accessing the sheetnames attribute. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> wb.sheetnames  # The workbook's sheets' names.
['Sheet1', 'Sheet2', 'Sheet3']
>>> sheet = wb['Sheet3']  # Get a sheet from the workbook.
>>> sheet
<Worksheet "Sheet3">
>>> type(sheet)
<class 'openpyxl.worksheet.worksheet.Worksheet'>
>>> sheet.title  # Get the sheet's title as a string.
'Sheet3'
>>> another_sheet = wb.active  # Get the active sheet.
>>> another_sheet
<Worksheet "Sheet1">
```

Each sheet is represented by a Worksheet object, which you can obtain by using the square brackets with the sheet name string, like a dictionary key. Finally, you can use the active attribute of a Workbook object to get the workbook's active sheet. The active sheet is the sheet that's on top when the workbook is opened in Excel. Once you have the Worksheet object, you can get its name from the title attribute.

Getting Cells from the Sheets

Once you have a Worksheet object, you can access a Cell object by its name. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']  # Get a sheet from the workbook.
>>> sheet['A1']  # Get a cell from the sheet.
<Cell 'Sheet1'.A1>
>>> sheet['A1'].value  # Get the value from the cell.
datetime.datetime(2035, 4, 5, 13, 34, 2)
>>> c = sheet['B1']  # Get another cell from the sheet.
>>> c.value
'Apples'
>>>  # Get the row, column, and value from the cell.
>>> f'Row {c.row}, Column {c.column} is {c.value}'
'Row 1, Column 2 is Apples'
>>> f'Cell {c.coordinate} is {c.value}'
'Cell B1 is Apples'
>>> sheet['C1'].value
73
```

The Cell object has a value attribute that contains, unsurprisingly, the value stored in that cell. It also has row, column, and coordinate attributes that provide location information for the cell. Here, accessing the value attribute of our Cell object for cell B1 gives us the string 'Apples'. The row attribute gives us the integer 1, the column attribute gives us 2, and the coordinate attribute gives us 'B1'.

The openpyxl module will automatically interpret the dates in column A and return them as datetime values rather than strings. Chapter 19 explains the datetime data type further.

Specifying a column by letter can be tricky to program, especially because after column Z, the columns start using two letters: AA, AB, AC, and so on. As an alternative, you can also get a cell using the sheet's cell() method and passing integers for its row and column keyword arguments. The first row or column integer is 1, not 0. Continue the interactive shell example by entering the following:

```
>>> sheet.cell(row=1, column=2)
<Cell 'Sheet1'.B1>
>>> sheet.cell(row=1, column=2).value
'Apples'
>>> for i in range(1, 8, 2): # Go through every other row.
...     print(i, sheet.cell(row=i, column=2).value)
...
1 Apples
3 Pears
5 Apples
7 Strawberries
```

Using the sheet's cell() method and passing it row=1 and column=2 gets you a Cell object for cell B1, just like specifying sheet['B1'] did.

By using this cell() method and its keyword arguments, we wrote a for loop to print the values of a series of cells. Say you want to go down column B and print the value in every cell with an odd row number. By passing 2 for the range() function's "step" parameter, you can get cells from every second row (in this case, all the odd-numbered rows). This example passes the for loop's i variable as the cell() method's row keyword, and uses 2 for the column keyword argument on each call of the method. Note that this method accepts the integer 2, not the string 'B'.

You can determine the size of the sheet with the Worksheet object's max_row and max_column attributes. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet.max_row  # Get the highest row number.
7
>>> sheet.max_column  # Get the highest column number.
3
```

Note that the max_column attribute is an integer rather than the letter that appears in Excel.

Converting Between Column Letters and Numbers

To convert from numbers to letters, call the `openpyxl.utils.get_column_letter()` function. To convert from letters to numbers, call the `openpyxl.utils.column_index_from_string()` function. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> from openpyxl.utils import get_column_letter, column_index_from_string
>>> get_column_letter(1)  # Translate column 1 to a letter.
'A'
>>> get_column_letter(2)
'B'
>>> get_column_letter(27)
'AA'
>>> get_column_letter(900)
'AHP'
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> get_column_letter(sheet.max_column)
'C'
>>> column_index_from_string('A') # Get A's number.
1
>>> column_index_from_string('AA')
27
```

After you import these two functions from the `openpyxl.utils` module, you can call `get_column_letter()` and pass it an integer like `27` to figure out what the letter name of the 27th column is. The function `column_index_string()` does the reverse: you pass it the letter name of a column, and it tells you what number that column is. You don't need to have a workbook loaded to use these functions.

Getting Rows and Columns

You can slice `Worksheet` objects to get all the `Cell` objects in a row, column, or rectangular area of the spreadsheet. Then, you can loop over all the cells in the slice. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet['A1':'C3']  # Get cells A1 to C3.
((<Cell 'Sheet1'.A1>, <Cell 'Sheet1'.B1>, <Cell 'Sheet1'.C1>), (<Cell 'Sheet1'.A2>, <Cell
'Sheet1'.B2>, <Cell 'Sheet1'.C2>), (<Cell 'Sheet1'.A3>, <Cell 'Sheet1'.B3>, <Cell
'Sheet1'.C3>))
>>> for row_of_cell_objects in sheet['A1':'C3']: ❶
...     for cell_obj in row_of_cell_objects: ❷
...         print(cell_obj.coordinate, cell_obj.value)
...     print('--- END OF ROW ---')
...
A1 2035-04-05 13:34:02
B1 Apples
C1 73
```

```
--- END OF ROW ---
A2 2035-04-05 03:41:23
B2 Cherries
C2 85
--- END OF ROW ---
A3 2035-04-06 12:46:51
B3 Pears
C3 14
--- END OF ROW ---
```

Here, we specify ['A1':'C3'] to get a slice of the Cell objects in the rectangular area from A1 to C3, and we get a tuple containing the Cell objects in that area.

This tuple contains three tuples: one for each row, from the top of the desired area to the bottom. Each of these three inner tuples contains the Cell objects in one row of our desired area, from the leftmost cell to the rightmost. So, overall, our slice of the sheet contains all the Cell objects in the area from A1 to C3, starting from the top-left cell and ending with the bottom-right cell.

To print the values of each cell in the area, we use two for loops. The outer for loop goes over each row in the slice ❶. Then, for each row, the nested for loop goes through each cell in that row ❷.

To access the values of cells in a particular row or column, you can also use a Worksheet object's rows and columns attributes. These attributes must be converted to lists with the list() function before you can use the square brackets and an index with them. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> list(sheet.columns)[1]  # Get the second column's cells.
(<Cell 'Sheet1'.B1>, <Cell 'Sheet1'.B2>, <Cell 'Sheet1'.B3>, <Cell 'Sheet1'.B4>, <Cell
'Sheet1'.B5>, <Cell 'Sheet1'.B6>, <Cell 'Sheet1'.B7>)
>>> for cell_obj in list(sheet.columns)[1]:
...     print(cell_obj.value)
...
Apples
Cherries
Pears
Oranges
Apples
Bananas
Strawberries
```

Using the rows attribute on a Worksheet object, passed to list(), will give us a list of tuples. Each of these tuples represents a row and contains the Cell objects in that row. The columns attribute, passed to list(), also gives us a list of tuples, with each of the tuples containing the Cell objects in a particular column. For *example3.xlsx*, because there are seven rows and three columns, list(sheet.rows) gives us a list of seven tuples (each containing three `Cell` objects), and `list(sheet.columns)` gives us a list of three tuples (each containing seven `Cell` objects).

To access one particular tuple, you can refer to it by its index in the larger tuple. For example, to get the tuple that represents column B, you'd use `list(sheet.columns)[1]`. To get the tuple containing the `Cell` objects in column A, you'd use `list(sheet.columns)[0]`. Once you have a tuple representing one row or column, you can loop through its `Cell` objects and print their values.

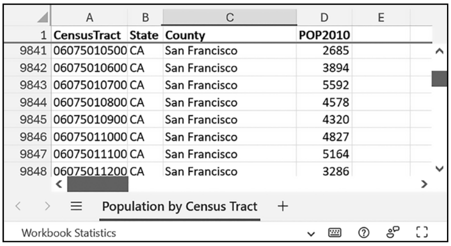
A REVIEW OF WORKBOOKS, SHEETS, AND CELLS

As a quick review, here's a rundown of all the functions, methods, and data types involved in reading a cell out of a spreadsheet file:

1. Import the `openpyxl` module.
2. Call the `openpyxl.load_workbook()` function to get a `Workbook` object.
3. Use the `active` or `sheetnames` attribute.
4. Get a `Worksheet` object.
5. Use indexing or the `cell()` sheet method with `row` and `column` keyword arguments.
6. Get a `Cell` object.
7. Read the `Cell` object's `value` attribute.

Project 9: Gather Census Statistics

Say you have a spreadsheet of data from the 2010 US Census and you've been given the boring task of going through its thousands of rows to count both the population and the number of census tracts for each county. (A *census tract* is simply a geographic area defined for the purposes of the census.) Each row represents a single census tract. We'll name the spreadsheet file *censuspopdata.xlsx*, and you can download it from this book's online resources. Its contents look like Figure 14-2.



	A	B	C	D	E
1	CensusTract	State	County	POP2010	
9841	06075010500	CA	San Francisco	2685	
9842	06075010600	CA	San Francisco	3894	
9843	06075010700	CA	San Francisco	5592	
9844	06075010800	CA	San Francisco	4578	
9845	06075010900	CA	San Francisco	4320	
9846	06075011000	CA	San Francisco	4827	
9847	06075011100	CA	San Francisco	5164	
9848	06075011200	CA	San Francisco	3286	

Figure 14-2: The censuspopdata.xlsx *spreadsheet*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Even though Excel can automatically calculate the sum of multiple selected cells, you'd still have to first manually select the cells for each of the 3,000-plus counties. Even if it takes just a few seconds to calculate a county's population by hand, this would take hours to do for the whole spreadsheet.

In this project, you'll write a script that can read from the census spreadsheet file and calculate statistics for each county in a matter of seconds.

This is what your program does:

- Reads the data from the Excel spreadsheet
- Counts the number of census tracts in each county
- Counts the total population of each county
- Prints the results

This means your code will need to do the following:

- Open and read the cells of an Excel document with the `openpyxl` module.
- Calculate all the tract and population data and store it in a data structure.
- Write the data structure to a text file with the *.py* extension using the `pprint` module so that it can be imported later.

Step 1: Read the Spreadsheet Data

There is just one sheet in the *censuspopdata.xlsx* spreadsheet, named `'Population by Census Tract'`, and each row in the sheet holds the data for a single census tract. The columns are the tract number (A), the state abbreviation (B), the county name (C), and the population of the tract (D).

Open a new file editor tab and enter the following code, then save the file as *readCensusExcel.py*:

```
  # readCensusExcel.py - Tabulates county population and census tracts

❶ import openpyxl, pprint
  print('Opening workbook...')
❷ wb = openpyxl.load_workbook('censuspopdata.xlsx')
❸ sheet = wb['Population by Census Tract']
  county_data = {}

  # TODO: Fill in county_data with each county's population and tracts.
  print('Reading rows...')
❹ for row in range(2, sheet.max_row + 1):
      # Each row in the spreadsheet has data for one census tract.
      state  = sheet['B' + str(row)].value
      county = sheet['C' + str(row)].value
      pop    = sheet['D' + str(row)].value

  # TODO: Open a new text file and write the contents of county_data to it.
```

This code imports the `openpyxl` module, as well as the `pprint` module that you'll use to print the final county data ❶. Then, it opens the *censuspopdata.xlsx* file ❷, gets the sheet with the census data ❸, and begins iterating over its rows ❹.

Note that you've also created a variable named `county_data`, which will contain the populations and number of tracts you calculate for each county. Before you can store anything in it, though, you should determine exactly how you'll structure the data inside it.

Step 2: Populate the Data Structure

In the United States, states have two-letter abbreviations and are further split into counties. The data structure stored in `county_data` will be a dictionary with state abbreviations as its keys. Each state abbreviation will map to another dictionary, whose keys are strings of the county names in that state. Each county name will in turn map to a dictionary with just two keys, `'tracts'` and `'pop'`. These keys map to the number of census tracts and the population for the county. For example, the dictionary will look similar to this:

```
{'AK': {'Aleutians East': {'pop': 3141, 'tracts': 1},
        'Aleutians West': {'pop': 5561, 'tracts': 2},
        'Anchorage': {'pop': 291826, 'tracts': 55},
        'Bethel': {'pop': 17013, 'tracts': 3},
        'Bristol Bay': {'pop': 997, 'tracts': 1},
        --snip--
```

If the previous dictionary were stored in `county_data`, the following expressions would evaluate like this:

```
>>> county_data['AK']['Anchorage']['pop']
291826
>>> county_data['AK']['Anchorage']['tracts']
55
```

More generally, the `county_data` dictionary's keys will look like this:

```
county_data[state abbrev][county]['tracts']
county_data[state abbrev][county]['pop']
```

Now that you know how `county_data` will be structured, you can write the code that will fill it with the county data. Add the following code to the bottom of your program:

```
# readCensusExcel.py - Tabulates county population and census tracts

--snip--

for row in range(2, sheet.max_row + 1):
    # Each row in the spreadsheet has data for one census tract.
    state  = sheet['B' + str(row)].value
    county = sheet['C' + str(row)].value
    pop    = sheet['D' + str(row)].value
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    # Make sure the key for this state exists.
❶   county_data.setdefault(state, {})
    # Make sure the key for this county in this state exists.
❷   county_data[state].setdefault(county, {'tracts': 0, 'pop': 0})

    # Each row represents one census tract, so increment by one.
❸   county_data[state][county]['tracts'] += 1
    # Increase the county pop by the pop in this census tract.
❹   county_data[state][county]['pop'] += int(pop)

# TODO: Open a new text file and write the contents of county_data to it.
```

The last two lines of code perform the actual calculation work, incrementing the value for `tracts` ❸ and increasing the value for `pop` ❹ for the current county on each iteration of the `for` loop.

The other code is there because you cannot add a county dictionary as the value for a state abbreviation key until the key itself exists in `county_data`. (That is, `county_data['AK']['Anchorage']['tracts'] += 1` will cause an error if the `'AK'` key doesn't exist yet.) To make sure the state abbreviation key exists in your data structure, you need to call the `setdefault()` method to set a value if one does not already exist for `state` ❶.

Just as the `county_data` dictionary needs a dictionary as the value for each state abbreviation key, each of *those* dictionaries will need its own dictionary as the value for each county key ❷. And each of *those* dictionaries in turn will need the keys `'tracts'` and `'pop'` that start with the integer value `0`. (If you ever lose track of the dictionary structure, look back at the example dictionary at the start of this section.)

Since `setdefault()` will do nothing if the key already exists, you can call it on every iteration of the `for` loop without a problem.

Step 3: Write the Results to a File

After the `for` loop has finished, the `county_data` dictionary will contain all of the population and tract information keyed by county and state. At this point, you could program more code to write this data to a text file or another Excel spreadsheet. For now, let's just use the `pprint.pformat()` function to write the `county_data` dictionary value as a massive string to a file named *census2010.py*. Add the following code to the bottom of your program (making sure to keep it un-indented so that it stays outside the `for` loop):

```
# readCensusExcel.py - Tabulates county population and census tracts.

--snip--

# Open a new text file and write the contents of county_data to it.
print('Writing results...')
result_file = open('census2010.py', 'w')
result_file.write('allData = ' + pprint.pformat(county_data))
result_file.close()
print('Done.')
```

The `pprint.pformat()` function produces a string that itself is formatted as valid Python code. By outputting it to a text file named *census2010.py*, you've generated a Python program from your Python program! This may seem complicated, but the advantage is that you can now import *census2010.py* just like any other Python module. In the interactive shell, change the current working directory to the folder with your newly created *census2010.py* file and then import it:

```
>>> import census2010
>>> census2010.allData['AK']['Anchorage']
{'pop': 291826, 'tracts': 55}
>>> anchorage_pop = census2010.allData['AK']['Anchorage']['pop']
>>> print('The 2010 population of Anchorage was ' + str(anchorage_pop))
The 2010 population of Anchorage was 291826
```

The *readCensusExcel.py* program was throwaway code: once you have its results saved to *census2010.py*, you won't need to run the program again. Whenever you need the county data, you can just run `import census2010`.

Calculating this data by hand would have taken hours; this program did it in a few seconds. Using `openpyxl`, you'll have no trouble extracting information saved to an Excel spreadsheet and performing calculations on it. You can download the complete program from the book's online resources.

Ideas for Similar Programs

Many businesses and offices use Excel to store various types of data, and it's not uncommon for spreadsheets to become large and unwieldy. Any program that parses an Excel spreadsheet has a similar structure: it loads the spreadsheet file, preps some variables or data structures, and then loops through each of the rows in the spreadsheet. Such a program could do the following:

- Compare data across multiple rows in a spreadsheet.
- Open multiple Excel files and compare data between spreadsheets.
- Check whether a spreadsheet has blank rows or invalid data in any cells and alert the user if it does.
- Read data from a spreadsheet and use it as the input for your Python programs.

Writing Excel Documents

The `openpyxl` module also provides ways of writing data, meaning that your programs can create and edit spreadsheet files. With Python, creating spreadsheets with thousands of rows of data is simple.

Creating and Saving Excel Files

Call the openpyxl.Workbook() function to create a new, blank Workbook object. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()  # Create a blank workbook.
>>> wb.sheetnames  # The workbook starts with one sheet.
['Sheet']
>>> sheet = wb.active
>>> sheet.title
'Sheet'
>>> sheet.title = 'Spam Bacon Eggs Sheet'  # Change the title.
>>> wb.sheetnames
['Spam Bacon Eggs Sheet']
```

The workbook will start off with a single sheet named *Sheet*. You can change the name of the sheet by storing a new string in its title attribute.

Anytime you modify the Workbook object or its sheets and cells, the spreadsheet file will not be saved until you call the save() workbook method. Enter the following into the interactive shell (with *example3.xlsx* in the current working directory):

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('example3.xlsx')
>>> sheet = wb['Sheet1']
>>> sheet.title = 'Spam Spam Spam'
>>> wb.save('example3_copy.xlsx')  # Save the workbook.
```

Here, we change the name of our sheet. To save our changes, we pass a filename as a string to the save() method. Passing a different filename than the original, such as 'example3_copy.xlsx', saves the changes to a copy of the spreadsheet.

Whenever you edit a spreadsheet you've loaded from a file, you should always save the new, edited spreadsheet with a different filename than the original. That way, you'll still have the original spreadsheet file to work with in case a bug in your code caused the new, saved file to contain incorrect or corrupted data. Also, the save() method won't work if the spreadsheet is currently open in the Excel desktop application. You must first close the spreadsheet and then run your Python program.

Creating and Removing Sheets

You can create or delete sheets from a workbook with the create_sheet() method and del operator. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> wb.sheetnames
['Sheet']
>>> wb.create_sheet()  # Add a new sheet.
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
<Worksheet "Sheet1">
>>> wb.sheetnames
['Sheet', 'Sheet1']
>>> # Create a new sheet at index 0.
>>> wb.create_sheet(index=0, title='First Sheet')
<Worksheet "First Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Sheet1']
>>> wb.create_sheet(index=2, title='Middle Sheet')
<Worksheet "Middle Sheet">
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
```

The `create_sheet()` method returns a new `Worksheet` object named `Sheet`*`X`*, which by default is the last sheet in the workbook. Optionally, you can specify the index and name of the new sheet with the `index` and `title` keyword arguments.

Continue the previous example by entering the following:

```
>>> wb.sheetnames
['First Sheet', 'Sheet', 'Middle Sheet', 'Sheet1']
>>> del wb['Middle Sheet']
>>> del wb['Sheet1']
>>> wb.sheetnames
['First Sheet', 'Sheet']
```

You can use the `del` operator to delete a sheet from a workbook, just like you can use it to delete a key-value pair from a dictionary.

Remember to call the `save()` method to save the changes after adding sheets to or removing sheets from the workbook.

Writing Values to Cells

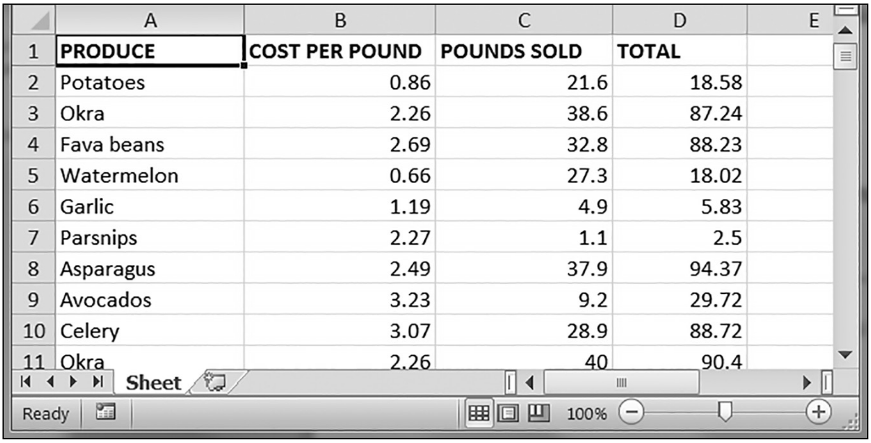
Writing values to cells is much like writing values to keys in a dictionary. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Hello, world!'  # Edit the cell's value.
>>> sheet['A1'].value
'Hello, world!'
```

If you have the cell's coordinate as a string, you can use it just like a dictionary key on the `Worksheet` object to specify which cell to write to.

Project 10: Update a Spreadsheet

In this project, you'll write a program to update cells in a spreadsheet of produce sales. Your program will look through the spreadsheet, find specific kinds of produce, and update their prices. Download this *produceSales3.xlsx* spreadsheet from the book's online resources. Figure 14-3 shows what the spreadsheet looks like.



	A	B	C	D	E
1	**PRODUCE**	**COST PER POUND**	**POUNDS SOLD**	**TOTAL**	
2	Potatoes	0.86	21.6	18.58	
3	Okra	2.26	38.6	87.24	
4	Fava beans	2.69	32.8	88.23	
5	Watermelon	0.66	27.3	18.02	
6	Garlic	1.19	4.9	5.83	
7	Parsnips	2.27	1.1	2.5	
8	Asparagus	2.49	37.9	94.37	
9	Avocados	3.23	9.2	29.72	
10	Celery	3.07	28.9	88.72	
11	Okra	2.26	40	90.4	

Figure 14-3: A spreadsheet of produce sales

Each row represents an individual sale. The columns are the type of produce sold (A), the cost per pound of that produce (B), the number of pounds sold (C), and the total revenue from the sale (D). The TOTAL column is set to an Excel formula like *=ROUND(B2*C2, 2)*, which multiplies the row's cost per pound by the number of pounds sold and rounds the result to the nearest cent. With this formula, the cells in the TOTAL column will automatically update themselves if there is a change in the COST PER POUND and POUNDS SOLD columns.

Now imagine that the prices of garlic, celery, and lemons were entered incorrectly, leaving you with the boring task of going through thousands of rows in this spreadsheet to update the cost per pound for any celery, garlic, and lemon rows. You can't do a simple find-and-replace for the price, because there might be other items with the same price that you don't want to mistakenly "correct." For thousands of rows, this would take hours to do by hand. But you can write a program that can accomplish this in seconds.

Your program should do the following:

- Loop over all the rows.
- If the row is for celery, garlic, or lemons, change the price.

This means your code will need to do the following:

- Open the spreadsheet file.
- For each row, check whether the value in column A is `Celery`, `Garlic`, or `Lemon`.
- If it is, update the price in column B.
- Save the spreadsheet to a new file (so that you don't lose the original spreadsheet, just in case).

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Step 1: Set Up a Data Structure with the Updated Information

The prices that you need to update are as follows:

- Celery: 1.19
- Garlic: 3.07
- Lemon: 1.27

You could write code to set these new prices, like this:

```
if produce_name == 'Celery':
    cell_obj = 1.19
if produce_name == 'Garlic':
    cell_obj = 3.07
if produce_name == 'Lemon':
    cell_obj = 1.27
```

But hardcoding the produce and updated price data like this is a bit inelegant. If you needed to update the spreadsheet again with different prices or different produce, you would have to change a lot of the code. Every time you change code, you risk introducing bugs.

A more flexible solution is to store the corrected price information in a dictionary and write your code to use this data structure. In a new file editor tab, enter the following code:

```
# updateProduce.py - Corrects costs in produce sales spreadsheet

import openpyxl

wb = openpyxl.load_workbook('produceSales3.xlsx')
sheet = wb['Sheet']

# The produce types and their updated prices
PRICE_UPDATES = {'Garlic': 3.07,
                 'Celery': 1.19,
                 'Lemon': 1.27}

# TODO: Loop through the rows and update the prices.
```

Save this as *updateProduce.py*. If you need to update the spreadsheet again, you'll need to update only the PRICE_UPDATES dictionary, not any other code.

Step 2: Check All Rows and Update Incorrect Prices

The next part of the program will loop through all the rows in the spreadsheet. Add the following code to the bottom of *updateProduce.py*:

```
#! python3
# updateProduce.py - Corrects costs in produce sales spreadsheet

--snip--
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
  # Loop through the rows and update the prices.
❶ for row_num in range(2, sheet.max_row + 1):  # Skip the first row.
    ❷ produce_name = sheet.cell(row=row_num, column=1).value
    ❸ if produce_name in PRICE_UPDATES:
          sheet.cell(row=row_num, column=2).value = PRICE_UPDATES[produce_name]

❹ wb.save('updatedProduceSales3.xlsx')
```

We loop through the rows starting at row 2, as row 1 is just the header ❶. The cell in column 1 (that is, column A) will be stored in the variable `produce_name` ❷. If `produce_name` exists as a key in the `PRICE_UPDATES` dictionary ❸, you know this row needs its price corrected. The correct price will be in `PRICE_UPDATES[produce_name]`.

Notice how clean using `PRICE_UPDATES` makes the code. It uses only one `if` statement, rather than a separate line like `if produce_name == 'Garlic':` for every type of produce to update. And since the code uses the `PRICE_UPDATES` dictionary instead of hardcoding the produce names and updated costs into the `for` loop, you can modify only the `PRICE_UPDATES` dictionary, and not the rest of the code, if the produce sales spreadsheet needs additional changes.

After going through the entire spreadsheet and making changes, the code saves the `Workbook` object to *updatedProduceSales.xlsx* ❹. It doesn't overwrite the old spreadsheet, in case there's a bug in the program and the updated spreadsheet is wrong. After checking that the updated spreadsheet looks right, you can delete the old spreadsheet.

Ideas for Similar Programs

Since many office workers use Excel spreadsheets all the time, a program that can automatically edit and write Excel files could be really useful. Such a program could do the following:

- Read data from one spreadsheet and write it to parts of other spreadsheets.
- Read data from websites, text files, or the clipboard and write it to a spreadsheet.
- Automatically "clean up" data in spreadsheets. For example, it could use regular expressions to read multiple formats of phone numbers and edit them to a single, standard format.

Setting the Font Style of Cells

Styling certain cells, rows, or columns can help you emphasize important areas in your spreadsheet. In the produce spreadsheet, for example, your program could apply bold text to the potato, garlic, and parsnip rows. Or perhaps you want to italicize every row with a cost per pound greater than $5. Styling parts of a large spreadsheet by hand would be tedious, but your programs can do it instantly.

To customize font styles in cells, import the Font() function from the openpyxl.styles module:

```
from openpyxl.styles import Font
```

Importing the function in this way allows you to write Font() instead of openpyxl.styles.Font(). (See "Importing Modules" on page XX in Chapter 3 for more information.)

The following example creates a new workbook and sets cell A1 to a 24-point italic font:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
❶ >>> italic_24_font = Font(size=24, italic=True)
❷ >>> sheet['A1'].font = italic_24_font
>>> sheet['A1'] = 'Hello, world!'
>>> wb.save('styles3.xlsx')
```

In this example, Font(size=24, italic=True) returns a Font object, which we store in italic_24_font ❶. The keyword arguments to Font() configure the object's styling information, and assigning sheet['A1'].font the italic_24_font object ❷ applies all of that font-styling information to cell A1.

To set font attributes, pass keyword arguments to Font(). Table 14-2 shows the possible keyword arguments for the Font() function.

Table 14-2: Keyword Arguments for Font Objects

Keyword argument	Data type	Description
name	String	The font name, such as 'Calibri' or 'Times New Roman'
size	Integer	The point size
bold	Boolean	True, for bold font
italic	Boolean	True, for italic font

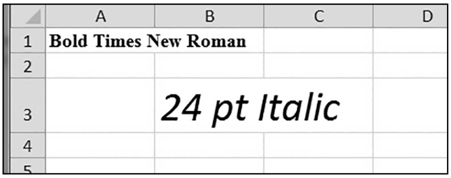
You can call Font() to create a Font object and store that Font object in a variable. You then assign that variable to a Cell object's font attribute. For example, this code creates various font styles:

```
>>> import openpyxl
>>> from openpyxl.styles import Font
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> bold_font = Font(name='Times New Roman', bold=True)
>>> sheet['A1'].font = bold_font
>>> sheet['A1'] = 'Bold Times New Roman'
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> italic_font = Font(size=24, italic=True)
>>> sheet['B3'].font = italic_font
>>> sheet['B3'] = '24 pt Italic'

>>> wb.save('styles3.xlsx')
```

Here, we store a `Font` object in `bold_font` and then set the A1 `Cell` object's `font` attribute to `bold_font`. We repeat the process with another `Font` object to set the font of a second cell. After you run this code, the styles of the A1 and B3 cells in the spreadsheet will have custom font styles, as shown in Figure 14-4.



	A	B	C	D
1	**Bold Times New Roman**			
2				
3		*24 pt Italic*		
4				
5				

Figure 14-4: A spreadsheet with custom font styles

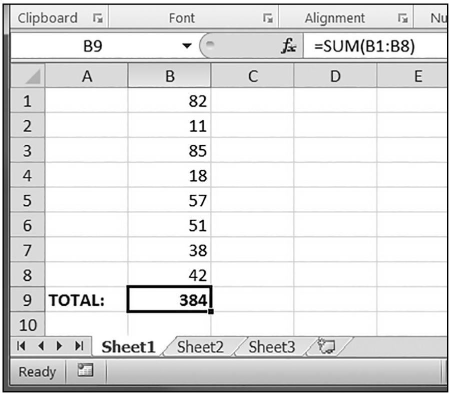
For cell A1, we set the font name to `'Times New Roman'` and set `bold` to `true`, so the text appears in bold Times New Roman. We didn't specify a point size, so the text uses the `openpyxl` default, 11. In cell B3, the text is italic, with a point size of 24. We didn't specify a font name, so the text uses the `openpyxl` default, Calibri.

Formulas

Excel formulas, which begin with an equal sign, can configure cells to contain values calculated from other cells. In this section, you'll use the `openpyxl` module to programmatically add formulas to cells, just as you would add any normal value. Here is an example:

```
>>> sheet['B9'] = '=SUM(B1:B8)'
```

This code will store the formula *=SUM(B1:B8)* in B9, setting the cell's value to the sum of values in cells B1 to B8. You can see this in action in Figure 14-5.



B9 =SUM(B1:B8)

	A	B	C	D	E
1		82			
2		11			
3		85			
4		18			
5		57			
6		51			
7		38			
8		42			
9	TOTAL:	384			
10					

Figure 14-5: Cell B9 contains the formula =SUM(B1:B8)*, which adds the values in cells B1 to B8.*

You can set an Excel formula just like any other text value in a cell. For instance, enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 200
>>> sheet['A2'] = 300
>>> sheet['A3'] = '=SUM(A1:A2)' # Set the formula.
>>> wb.save('writeFormula3.xlsx')
```

The cells in A1 and A2 are set to 200 and 300, respectively. The value in cell A3 is set to a formula that sums the values in A1 and A2. When the spreadsheet is opened in Excel, A3 will display its value as 500.

The `openpyxl` module doesn't have the ability to calculate Excel formulas and populate cells with the results. However, if you open this *writeFormula3.xlsx* file in Excel, Excel itself will populate the cells with the formula results. You can save the file in Excel, and then open it while passing the `data_only=True` keyword argument to `openpyxl.load_workbook()`, and the cell values should show the calculation results instead of the formula string:

```
>>> # Be sure to open writeFormula3.xlsx in Excel and save it first.
>>> import openpyxl
>>> wb = openpyxl.load_workbook('writeFormula3.xlsx')  # Open without data_only.
>>> wb.active['A3'].value  # Get the formula string.
'=SUM(A1:A2)'
>>> wb = openpyxl.load_workbook('writeFormula3.xlsx', data_only=True)  # Open with data_only.
>>> wb.active['A3'].value  # Get the formula result.
500
```

Again, you'll only see the `500` result in the spreadsheet file if you opened and saved it in Excel so that Excel could run the formula calculation and store the result in the spreadsheet file. This is the value `openpyxl` reads when you pass `data_only=True` to `openpyxl.load_workbook()`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Excel formulas offer a level of programmability for spreadsheets, but they can quickly become unmanageable for complicated tasks. For example, even if you're deeply familiar with Excel formulas, it's a headache to try to decipher what *=IFERROR(TRIM(IF(LEN(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE))>0,SUBSTITUTE(VLOOKUP(F7, Sheet2!$A$1:$B$10000, 2, FALSE), " ", ""),"")), "")* actually does. Python code is much more readable.

Adjusting Rows and Columns

In Excel, adjusting the sizes of rows and columns is as easy as clicking and dragging the edges of a row or column header. But if you need to set the size of a row or column based on its cells' contents, or if you want to set sizes in a large number of spreadsheet files, it's much quicker to write a Python program to do it.

You can also hide rows and columns from view, or "freeze" them in place so that they're always visible on the screen, appearing on every page when you print the spreadsheet (which is handy for headers).

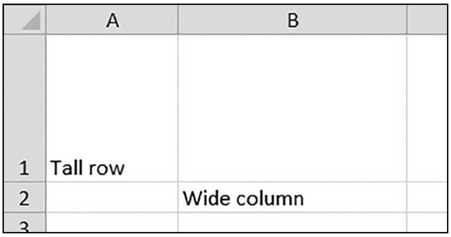
Setting Row Height and Column Width

A `Worksheet` object has `row_dimensions` and `column_dimensions` attributes that control row heights and column widths. For example, enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet['A1'] = 'Tall row'
>>> sheet['B2'] = 'Wide column'
>>> sheet.row_dimensions[1].height = 70
>>> sheet.column_dimensions['B'].width = 20
>>> wb.save('dimensions3.xlsx')
```

A sheet's `row_dimensions` and `column_dimensions` are dictionary-like values; `row_dimensions` contains `RowDimension` objects, and `column_dimensions` contains `ColumnDimension` objects. In `row_dimensions`, you can access one of the objects using the number of the row (in this case, 1 or 2). In `column_dimensions`, you can access one of the objects using the letter of the column (in this case, A or B).

The *dimensions3.xlsx* spreadsheet looks like Figure 14-6.



	A	B	
1	Tall row		
2		Wide column	
3			

Figure 14-6: Row 1 and column B set to larger heights and widths

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The default width and height of cells varies between versions of Excel and `openpyxl`.

Merging and Unmerging Cells

You can merge a rectangular group of cells into a single cell with the `merge_cells()` sheet method. Enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb['Sheet']
>>> sheet.merge_cells('A1:D3')  # Merge all these cells.
>>> sheet['A1'] = 'Twelve cells merged together.'
>>> sheet.merge_cells('C5:D5')  # Merge these two cells.
>>> sheet['C5'] = 'Two merged cells.'
>>> wb.save('merged3.xlsx')
```

The argument to `merge_cells()` is a single string of the top-left and bottom-right cells of the rectangular area to be merged: `'A1:D3'` merges 12 cells into a single cell. To set the value of these merged cells, simply set the value of the top-left cell of the merged group.

When you run this code, *merged.xlsx* will look like Figure 14-7.

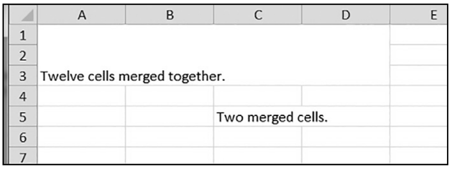


Figure 14-7: Merged cells in a spreadsheet

To unmerge cells, call the `unmerge_cells()` sheet method:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('merged3.xlsx')
>>> sheet = wb['Sheet']
>>> sheet.unmerge_cells('A1:D3')  # Split these cells up.
>>> sheet.unmerge_cells('C5:D5')
>>> wb.save('unmerged3.xlsx')
```

If you save your changes and then take a look at the spreadsheet, you'll see that the merged cells have gone back to being individual cells.

Freezing Panes

For spreadsheets that are too large to be displayed all at once, it's helpful to "freeze" a few of the top rows or leftmost columns onscreen. Frozen column or row headers, for example, are always visible to the user even as they scroll through the spreadsheet. These are known as *freeze panes*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

In openpyxl, each Worksheet object has a freeze_panes attribute that you can set to a Cell object or a string of a cell's coordinates. Note that this attribute will freeze all rows above this cell and all columns to the left of it, but not the row and column of the cell itself. To unfreeze all panes, set freeze_panes to None or 'A1'. Table 14-3 shows which rows and columns get frozen for some example settings of freeze_panes.

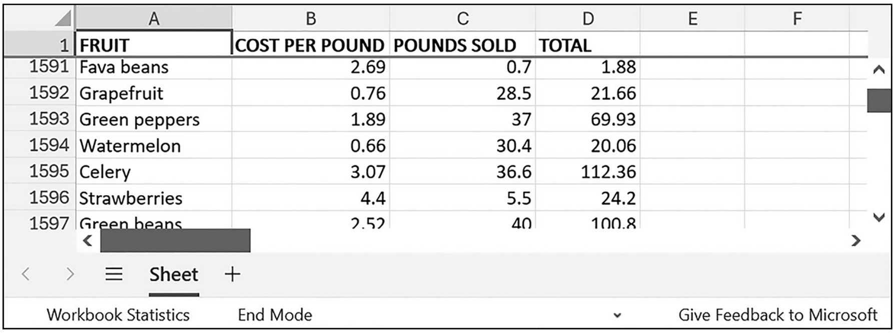
Table 14-3: Frozen Pane Examples

freeze_panes setting	Rows and columns frozen
`sheet.freeze_panes = 'A2'`	Row 1 (no columns frozen)
`sheet.freeze_panes = 'B1'`	Column A (no rows frozen)
`sheet.freeze_panes = 'C1'`	Columns A and B (no rows frozen)
`sheet.freeze_panes = 'C2'`	Row 1 and columns A and B
`sheet.freeze_panes = 'A1'` or `sheet.freeze_panes = None`	No frozen rows or columns

Download another copy of the *produceSales3.xlsx* spreadsheet, then enter the following into the interactive shell:

```
>>> import openpyxl
>>> wb = openpyxl.load_workbook('produceSales3.xlsx')
>>> sheet = wb.active
>>> sheet.freeze_panes = 'A2'  # Freeze the rows above A2.
>>> wb.save('freezeExample3.xlsx')
```

You can see the result in Figure 14-8.



	A	B	C	D	E	F
1	FRUIT	COST PER POUND	POUNDS SOLD	TOTAL		
1591	Fava beans	2.69	0.7	1.88		
1592	Grapefruit	0.76	28.5	21.66		
1593	Green peppers	1.89	37	69.93		
1594	Watermelon	0.66	30.4	20.06		
1595	Celery	3.07	36.6	112.36		
1596	Strawberries	4.4	5.5	24.2		
1597	Green beans	2.52	40	100.8		

Figure 14-8: Freezing row 1

Because you set the freeze_panes attribute to 'A2', row 1 will remain visible, no matter where the user scrolls in the spreadsheet.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Charts

The openpyxl module supports creating bar, line, scatter, and pie charts using the data in a sheet's cells. To make a chart, you need to do the following:

1. Create a Reference object from a rectangular selection of cells.
2. Create a Series object by passing in the Reference object.
3. Create a Chart object.
4. Append the Series object to the Chart object.
5. Add the Chart object to the Worksheet object, optionally specifying which cell should be the top-left corner of the chart.

The Reference object requires some explaining. To create Reference objects, you must call the openpyxl.chart.Reference() function and pass five arguments:

- The Worksheet object containing your chart data.
- The column and row integer of the top-left cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column. Note that 1 is the first row, not 0.
- The column and row integer of the bottom-right cell of the rectangular selection of cells containing your chart data: the first integer in the tuple is the row, and the second is the column.

Enter this interactive shell example to create a bar chart and add it to the spreadsheet:

```
>>> import openpyxl
>>> wb = openpyxl.Workbook()
>>> sheet = wb.active
>>> for i in range(1, 11):  # Create some data in column A.
...     sheet['A' + str(i)] = i * i
...
>>> ref_obj = openpyxl.chart.Reference(sheet, 1, 1,
1, 10)

>>> series_obj = openpyxl.chart.Series(ref_obj, title='First series')

>>> chart_obj = openpyxl.chart.BarChart()
>>> chart_obj.title = 'My Chart'
>>> chart_obj.append(series_obj)

>>> sheet.add_chart(chart_obj, 'C5')
>>> wb.save('sampleChart3.xlsx')
```

This code produces a spreadsheet that looks like Figure 14-9.

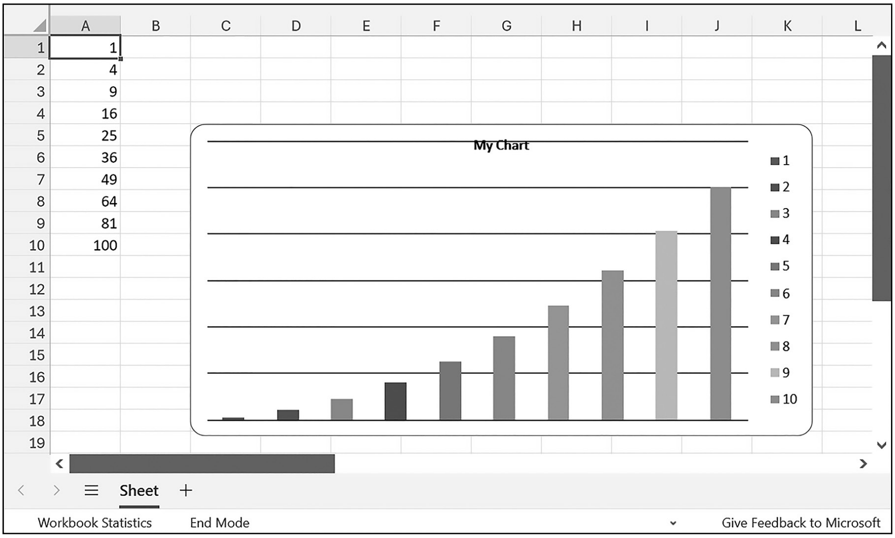


Figure 14-9: A spreadsheet with a chart added

We've created a bar chart by calling `openpyxl.chart.BarChart()`. You can also create line charts, scatter charts, and pie charts by calling `openpyxl.chart.LineChart()`, `openpyxl.chart.ScatterChart()`, and `openpyxl.chart.PieChart()`.

Summary

Often, the hard part of processing information isn't the processing itself but simply getting the data in the right format for your program. But once you've loaded your Excel spreadsheet into Python, you can extract and manipulate its data much faster than you could by hand.

You can also generate spreadsheets as output from your programs. So, if colleagues need a text file or PDF of thousands of sales contacts transferred to a spreadsheet file, you won't have to tediously copy and paste it all into Excel. Equipped with the `openpyxl` module and some programming knowledge, you'll find processing even the biggest spreadsheets a piece of cake.

In the next chapter, we'll take a look at using Python to interact with another spreadsheet program: the popular online Google Sheets application.

Practice Questions

For the following questions, imagine you have a `Workbook` object in the variable `wb`, a `Worksheet` object in `sheet`, and a `Sheet` object in `sheet`.

1. What does the `openpyxl.load_workbook()` function return?
2. What does the `wb.sheetnames` workbook attribute contain?
3. How would you retrieve the `Worksheet` object for a sheet named `'Sheet1'`?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

4. How would you retrieve the `Worksheet` object for the workbook's active sheet?
5. How would you retrieve the value in cell C5?
6. How would you set the value in cell C5 to `"Hello"`?
7. How would you retrieve the cell's row and column as integers?
8. What do the `sheet.max_column` and `sheet.max_row` sheet attributes hold, and what is the data type of these attributes?
9. If you needed to get the integer index for column `'M'`, what function would you need to call?
10. If you needed to get the string name for row `14`, what function would you need to call?
11. How can you retrieve a tuple of all the `Cell` objects from A1 to F1?
12. How would you save the workbook to the filename *example3.xlsx*?
13. How do you set a formula in a cell?
14. If you want to retrieve the result of a cell's formula instead of the cell's formula itself, what must you do first?
15. How would you set the height of row 5 to 100?
16. How would you hide column C?
17. What is a freeze pane?
18. What five functions and methods do you have to call to create a bar chart?

Practice Programs

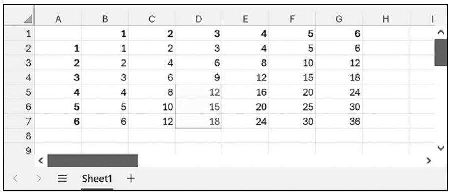
For practice, write programs to do the following tasks.

Multiplication Table Maker

Create a program *multiplicationTable.py* that takes a number *N* from the command line and creates an *N*×*N* multiplication table in an Excel spreadsheet. For example, when the program is run like this

```
py multiplicationTable.py 6
```

it should create a spreadsheet that looks like Figure 14-10.



	A	B	C	D	E	F	G	H	I
1		**1**	**2**	**3**	**4**	**5**	**6**		
2	**1**	1	2	3	4	5	6		
3	**2**	2	4	6	8	10	12		
4	**3**	3	6	9	12	15	18		
5	**4**	4	8	12	16	20	24		
6	**5**	5	10	15	20	25	30		
7	**6**	6	12	18	24	30	36		
8									
9									

Figure 14-10: A multiplication table generated in a spreadsheet

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

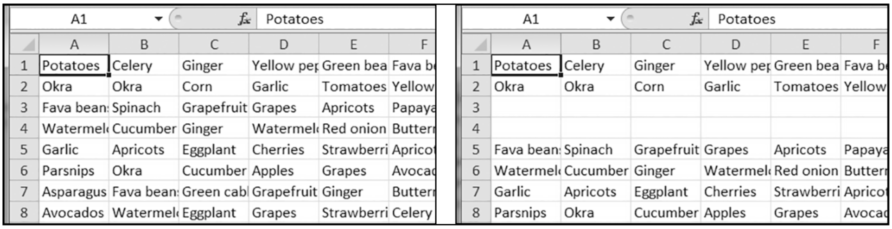
Row 1 and column A should contain labels and be in bold.

Blank Row Inserter

Create a program *blankRowInserter.py* that takes two integers and a filename string as command line arguments. Let's call the first integer *N* and the second integer *M*. Starting at row *N*, the program should insert *M* blank rows into the spreadsheet. For example, when the program is run like this . . .

```
python blankRowInserter.py 3 2 myProduce.xlsx
```

. . . the "before" and "after" spreadsheets should look like Figure 14-11.



	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3	Fava bean:	Spinach	Grapefruit	Grapes	Apricots	Papaya
4	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
5	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
6	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad
7	Asparagus	Fava bean:	Green cabl	Grapefruit	Ginger	Butterr
8	Avocados	Watermel	Eggplant	Grapes	Strawberri	Celery

	A	B	C	D	E	F
1	Potatoes	Celery	Ginger	Yellow pep	Green bea	Fava be
2	Okra	Okra	Corn	Garlic	Tomatoes	Yellow
3						
4						
5	Fava bean:	Spinach	Grapefruit	Grapes	Apricots	Papaya
6	Watermel	Cucumber	Ginger	Watermel	Red onion	Butterr
7	Garlic	Apricots	Eggplant	Cherries	Strawberri	Apricot
8	Parsnips	Okra	Cucumber	Apples	Grapes	Avocad

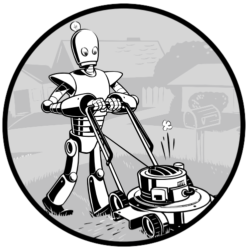
Figure 14-11: Before (left) and after (right) the two blank rows are inserted at row 3

You can write this program by reading in the contents of the spreadsheet. Then, when writing out the new spreadsheet, use a `for` loop to copy the first *N* lines. For the remaining lines, add *M* to the row number in the output spreadsheet.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

15

GOOGLE SHEETS



Google Sheets, the free, web-based spreadsheet application available to anyone with a Google account or Gmail address, has become a useful, feature-rich competitor to Excel. Google Sheets has its own API, but this API can be confusing to learn and use. This chapter covers the EZSheets third-party library, which presents you with a simpler way to perform common actions, handling the details of the Google Sheets API so that you don't have to learn them.

Installing and Setting Up EZSheets

You can install EZSheets with the pip command line tool by following the instructions in Appendix A.

Before your Python scripts can use EZSheets to access and edit your Google Sheets spreadsheets, you need a credentials JSON file and two token JSON files. There are five parts to creating credentials:

1. Create a new Google Cloud project.
2. Enable the Google Sheets API and Google Drive API for your project.
3. Configure the OAuth consent screen.
4. Create credentials.
5. Log in with the credentials file.

This may seem like a lot of work, but you have to perform this setup only once, and doing it is free. You'll need a Google/Gmail account; I strongly recommend creating a new Google account instead of using your existing one, to prevent a bug in your Python script from affecting the spreadsheets in your personal Google account. Throughout this chapter, I'll say *your Google account* and *your Gmail email address* to refer to the Google account that owns the spreadsheets your Python program accesses.

Google may slightly change the layout or wording on its Google Cloud Console website. However, the basic steps I've outlined should remain the same.

Creating a New Google Cloud Project

First, you need to set up a Google Cloud project. In your browser, go to *https://console.cloud.google.com* and sign in to your Google account with your username and password. You will be taken to a Getting Started page. At the top of the page, click **Select a project**. In the pop-up window that appears, click **New Project**. This should take you to a new project page.

Google Cloud will generate a project name like "My Project 23135" for you, along with a random project ID, like "macro-nuance-362516." These values won't be visible to users of your Python scripts, and you can change the project name to whatever name you want, but you cannot change the project ID. I just use the default name that the website generates for me. You can leave the location set to "No organization." Free Google accounts can have up to 12 projects, but you need only one project for all the Python scripts you want to create. Click the blue **Create** button to create the project.

Enabling the Sheets and Drive APIs

On the *https://console.cloud.google.com* page, click the **Navigation** button in the upper left. (The icon has three horizontal stripes and is often called the *hamburger* icon.) Go to **APIs & Services ▸ Library** to visit the API Library page. You'll see many Google APIs for Gmail, Google Maps, Google Cloud Storage, and other Google services. We need to allow our project to use the Google Sheets and Google Drive APIs. EZSheets uses the Google Drive API to upload and download spreadsheet files.

Scroll down, locate the Google Sheets API, and click it, or enter "Google Sheets API" into the search bar to find it. This should take you to the Google Sheets API page. Click the blue **Enable** button to enable your Google Cloud project to use the Google Sheets API. You'll be redirected to the **APIs & Services ▸ Enabled APIs & Services** page, where you can find information about how often your Python scripts are using this API. Repeat this process for the Google Drive API to enable it as well.

Next, you need to configure your project's OAuth consent screen.

Configuring the OAuth Consent Screen

The OAuth consent screen will appear to the user when they first run `import ezsheets`. On the *Step 1 OAuth consent screen* page, select **External** and click the blue **Create** button. The next page should show what the OAuth consent screen looks like. Pick a name for the App Name field (I use something generic, like "Python Google API Script"), and enter your email address for the User Support Email and Developer Contact Information field. Then click the **Save and Continue** button.

On the *Step 2 Scopes* page, define your project's scopes, or the permissions for the resources the project is allowed to access. Click the **Add or Remove Scopes** button, and in the new panel that appears, go through the table and check the checkboxes for the scopes *.../auth/drive* (the Google Drive API) and *.../auth/spreadsheets* (the Google Sheets API). Then, click the blue **Update** button and then click **Save and Continue**.

The *Step 3 Test users* page requires you to add the Gmail email addresses of the Google accounts that own the spreadsheets your Python script will interact with. Unless you go through Google's app approval process, your scripts are limited to interacting with the email addresses you provide in this step. Click the **+ Add Users** button. In the new panel that appears, enter the Gmail address of your Google account and click the blue **Add** button. Then click **Save and Continue**.

The *Step 4 Summary* page provides a summary of the previous steps. If all the information looks right, click the **BACK TO DASHBOARD** button. The next step is to create credentials for your project.

Creating Credentials

First, you'll need to create a credentials file. EZSheets needs this to use the Google API, even for spreadsheets that are publicly shared. From the Navigation sidebar menu, click **APIs & Services** and then **Credentials** to go to the Credentials page. Then click the **+ Create Credentials** link at the top of the page. A submenu should open asking what kind of credentials you want to create: API Key, OAuth Client ID, or Service Account. Click **OAuth Client ID**.

On the next page, select **Desktop App** for the Application Type and leave Name as the default "Desktop client 1." You can change it to a different name if you want; it doesn't appear to the users of your Python script. Click the blue **Create** button.

A pop-up window should appear. Click **Download JSON** to download the credentials file, which should have a name like *client_secret_282792235794-p2o9gfcub4htibfg2u207gcomco9nqm7.apps.googleusercontent.com.json*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Place it in the same folder as your Python script. For simplicity, you can also rename the JSON file to *credentials-sheets.json*. EZSheets searches for *credentials-sheets.json* or any file that matches the *client_secret_*.json* format.

Logging In with the Credentials File

Run the Python interactive shell from the same folder that the credentials JSON file is in and then run **`import ezsheets`**. EZSheets automatically checks the current working directory for the credentials JSON file by calling the `ezsheets.init()` function. If the file is found, EZSheets launches your web browser to the OAuth consent screen to generate token files. EZSheets also requires these token files, named *token-drive.pickle* and *token-sheets.pickle*, along with the credentials file to access spreadsheets. Generating token files is a one-time setup step that won't happen the next time you run `import ezsheets`.

Sign in with your Google account. This must be the same email address you provided for the "Test User" when configuring the Google Cloud project's OAuth consent screen. You should get a warning message that reads, "Google hasn't verified this app," which is fine, because you are the app creator. Click the **Continue** link. You should arrive at another page that says something like "Python Google API Script wants access to your Google Account" (or whichever name you gave in the OAuth consent screen setup). Click **Continue**. You'll come to a plain web page that says, "The authentication flow has completed." You can now close the browser window.

Once you've completed the authentication flow for the Sheets API, you must repeat this process for the Drive API in the window that opens next. After closing the second window, you should now see *token-drive.pickle* and *token-sheets.pickle* files in the same folder as your credentials JSON file. Treat these files like passwords and do not share them: they can be used to log in and access your Google Sheets spreadsheets.

Revoking the Credentials File

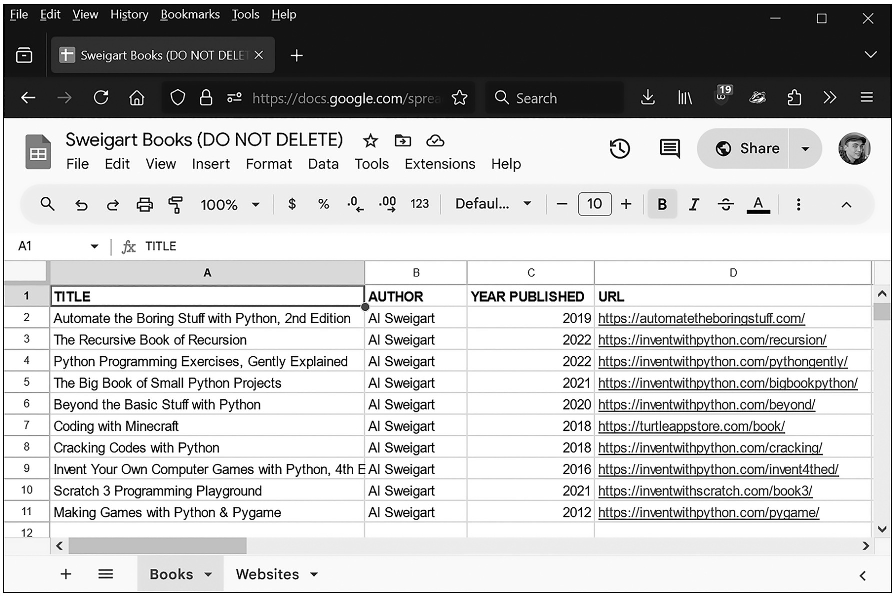
If you accidentally share the credential or token files with someone, they won't be able to change your Google account password, but they will have access to your spreadsheets. You can revoke these files by logging in to *https://console.developers.google.com*. Click the **Credentials** link on the sidebar. Then, in the OAuth 2.0 Client IDs table, click the trash can icon next to the credentials file you've accidentally shared. Once revoked, the credentials and token files are useless, and you can delete them. You'll then have to generate a new credentials JSON file and token files.

Spreadsheet Objects

In Google Sheets, a *spreadsheet* can contain multiple *sheets* (also called *worksheets*), and each sheet contains columns and rows of cells. Cells contain data such as numbers, dates, or bits of text. Cells also have properties such as fonts, widths and heights, and background colors. Figure 15-1

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

shows a spreadsheet titled *Sweigart Books* containing two sheets, titled *Books* and *Websites*. You can view this spreadsheet in your browser by going to *https://autbor.com/examplegs*. The first column of each sheet is labeled *A*, and the first row is labeled *1*. (This differs from Python lists, whose first item appears at index 0.)


File Edit View History Bookmarks Tools Help
Sweigart Books (DO NOT DELE
https://docs.google.com/sprea
Search
Sweigart Books (DO NOT DELETE)
File Edit View Insert Format Data Tools Extensions Help
Share
100%
$ % .0 .00 123
Defaul...
10
B I A
A1
TITLE
Books
Websites

	A	B	C	D
1	TITLE	AUTHOR	YEAR PUBLISHED	URL
2	Automate the Boring Stuff with Python, 2nd Edition	Al Sweigart	2019	https://automatetheboringstuff.com/
3	The Recursive Book of Recursion	Al Sweigart	2022	https://inventwithpython.com/recursion/
4	Python Programming Exercises, Gently Explained	Al Sweigart	2022	https://inventwithpython.com/pythongently/
5	The Big Book of Small Python Projects	Al Sweigart	2021	https://inventwithpython.com/bigbookpython/
6	Beyond the Basic Stuff with Python	Al Sweigart	2020	https://inventwithpython.com/beyond/
7	Coding with Minecraft	Al Sweigart	2018	https://turtleappstore.com/book/
8	Cracking Codes with Python	Al Sweigart	2018	https://inventwithpython.com/cracking/
9	Invent Your Own Computer Games with Python, 4th E	Al Sweigart	2016	https://inventwithpython.com/invent4thed/
10	Scratch 3 Programming Playground	Al Sweigart	2021	https://inventwithscratch.com/book3/
11	Making Games with Python & Pygame	Al Sweigart	2012	https://inventwithpython.com/pygame/
12				

Figure 15-1: A spreadsheet titled Sweigart Books *with two sheets,* Books *and* Websites

While most of your work will involve modifying the `Sheet` objects, you can also modify `Spreadsheet` objects, as you'll see in the next section.

Creating, Uploading, and Listing Spreadsheets

You can make a new `Spreadsheet` object from an existing Google Sheets spreadsheet, a new blank spreadsheet, or an uploaded Excel spreadsheet. All Google Sheets spreadsheets have a unique ID that can be found in their URL, after the *spreadsheets/d/* part and before the */edit* part. For example, in the URL *https://docs.google.com/spreadsheets/d/1TzOJxhNKr15tzdZxTqtQ3EmDP6em_elnbtmZIcyu8vI/edit#gid=0/*, the ID would be *1TzOJxhNKr15tzdZxTqtQ3EmDP6em_elnbtmZIcyu8vI*.

A Google Sheets spreadsheet is represented as an `ezsheets.Spreadsheet` object, which has `id`, `url`, and `title` attributes. You can create a new, blank spreadsheet with the `Spreadsheet()` function:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet()
>>> ss.title = 'Title of My New Spreadsheet'
>>> ss.title
'Title of My New Spreadsheet'
```

```
>>> ss.url
'https://docs.google.com/spreadsheets/d/1gxz-Qr2-RNtqi_d7wWlsDlbtPLRQigcEXvCtdVwmH40/'
>>> ss.id
'1gxz-Qr2-RNtqi_d7wWlsDlbtPLRQigcEXvCtdVwmH40'
```

You can also load an existing spreadsheet by passing its ID or URL, or a URL that redirects to its URL:

```
>>> import ezsheets
>>> ss1 = ezsheets.Spreadsheet('https://autbor.com/examplegs')
>>> ss2 = ezsheets.Spreadsheet('https://docs.google.com/spreadsheets/d/1TzOJxh
NKr15tzdZxTqtQ3EmDP6em_elnbtmZIcyu8vI/')
>>> ss3 = ezsheets.Spreadsheet('1TzOJxhNKr15tzdZxTqtQ3EmDP6em_elnbtmZIcyu8vI')
>>> ss1 == ss2 == ss3  # These are the same spreadsheet.
True
```

To upload an existing Excel, OpenOffice, CSV, or TSV spreadsheet to Google Sheets, pass the spreadsheet's filename to `ezsheets.upload()`. Enter the following into the interactive shell, replacing *my_spreadsheet.xlsx* with a spreadsheet file of your own:

```
>>> import ezsheets
>>> ss = ezsheets.upload('my_spreadsheet.xlsx')
>>> ss.title
'my_spreadsheet'
```

You can list the spreadsheets in your Google account by calling the `listSpreadsheets()` function. This function returns a dictionary whose keys are spreadsheet IDs and whose values are the titles of each spreadsheet. It includes deleted spreadsheets in your account's *Trash* folder. Try entering the following into the interactive shell after uploading a spreadsheet:

```
>>> ezsheets.listSpreadsheets()
{'1J-Jx6Ne2K_vqI9J2SO-TAXOFbxx_9tUjwnkPC22LjeU': 'Education Data'}
```

Once you've obtained a `Spreadsheet` object, you can use its attributes and methods to manipulate the online spreadsheet hosted on Google Sheets.

Accessing Spreadsheet Attributes

While the actual data lives in a spreadsheet's individual sheets, the `Spreadsheet` object has the following attributes for manipulating the spreadsheet itself: `title`, `id`, `url`, `sheetTitles`, and `sheets`. Let's examine the spreadsheet at *https://autbor.com/examplegs*. Your Google account has permissions to view but not modify it, but you can copy the sheet to a newly created spreadsheet in your own account:

```
>>> import ezsheets
>>> example_ss = ezsheets.Spreadsheet('https://autbor.com/examplegs')
>>> ss = ezsheets.Spreadsheet()
>>> example_ss.sheets[0].copyTo(ss)
>>> ss.sheets[0].delete()  # Delete the Sheet1 sheet.
>>> ss.url
'https://docs.google.com/spreadsheets/d/15gjrbgTmUzItRt9KUcL4JajLaQU7OxanstB1dXKoSlM/'
```

The newly copied sheet will have the title *Copy of Books*, as *Books* was the name of the original sheet. Continue the interactive shell example with the following code:

```
>>> ss.title  # The title of the spreadsheet
'Untitled spreadsheet'
>>> ss.title = 'Sweigart Books'  # Change the title.
>>> ss.id  # The unique ID (a read-only attribute)
'15gjrbgTmUzItRt9KUcL4JajLaQU7OxanstB1dXKoSlM'
>>> ss.url  # The original URL (a read-only attribute)
'https://docs.google.com/spreadsheets/d/15gjrbgTmUzItRt9KUcL4JajLaQU7OxanstB1dXKoSlM/'
>>> ss.sheetTitles  # The titles of all the Sheet objects
('Copy of Books',)
>>> ss.sheets  # The Sheet objects in this Spreadsheet, in order
(<Sheet sheetId=1464919459, title='Copy of Books', rowCount=1000, columnCount=26>,)
>>> ss.sheets[0]  # The first Sheet object in this Spreadsheet
<Sheet sheetId=1464919459, title='Copy of Books', rowCount=1000, columnCount=26>
>>> ss['Copy of Books']  # Sheets can also be accessed by title.
<Sheet sheetId=1464919459, title='Copy of Books', rowCount=1000, columnCount=26>
>>> ss.Sheet('New blank sheet')  # Create a new sheet.
<Sheet sheetId=1759616008, title='New blank sheet', rowCount=1000, columnCount=26>
>>> ss.sheets[1].delete()  # Delete the second Sheet object in this Spreadsheet.
```

If someone changes the spreadsheet in their browser, your script can update the `Spreadsheet` object to match the online data by calling the `refresh()` method:

```
>>> ss.refresh()
```

This will refresh not only the `Spreadsheet` object's attributes but also the data in the `Sheet` objects it contains. You'll see the changes you make to the `Spreadsheet` object applied to the online spreadsheet in real time.

Downloading and Uploading Spreadsheets

You can download a Google Sheets spreadsheet in a number of formats: Excel, OpenOffice, CSV, TSV, and PDF. You can also download it as a ZIP file containing HTML files of the spreadsheet's data. EZSheets contains functions for each of these options:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet('https://autbor.com/examplegs')
>>> ss.title
'Sweigart Books (DO NOT DELETE)'
>>> ss.downloadAsExcel()  # Downloads the spreadsheet as an Excel file
'Sweigart_Books.xlsx'
>>> ss.downloadAsODS()  # Downloads the spreadsheet as an OpenOffice file
'Sweigart_Books.ods'
```

```
>>> ss.downloadAsCSV()  # Downloads only the first sheet as a CSV file
'Sweigart_Books.csv'
>>> ss.downloadAsTSV()  # Downloads only the first sheet as a TSV file
'Sweigart_Books.tsv'
>>> ss.downloadAsPDF()  # Downloads the spreadsheet as a PDF
'Sweigart_Books.pdf'
>>> ss.downloadAsHTML()  # Downloads the spreadsheet as a ZIP of HTML files
'Sweigart_Books.zip'
```

Note that files in the CSV or TSV format can contain only one sheet; therefore, if you download a Google Sheets spreadsheet in either of these formats, you will get the first sheet only. To download other sheets, you'll need to reorder the Sheet objects before downloading.

The download functions all return a string of the downloaded file's filename. You can also specify your own filename for the spreadsheet by passing the new filename to the download function:

```
>>> ss.downloadAsExcel('a_different_filename.xlsx')
'a_different_filename.xlsx'
```

The function returns the local filename.

Deleting Spreadsheets

To delete a spreadsheet, call the delete() method:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet()  # Create the spreadsheet.
>>> ezsheets.listSpreadsheets()  # Confirm that we've created a spreadsheet.
{'1aCw2NNJSZblDbhygVv77kPsL3djmgV5zJZllSOZ_mRk': 'Delete me'}
>>> ss.delete()  # Delete the spreadsheet.
>>> ezsheets.listSpreadsheets()  # Spreadsheets in the Trash folder are still listed.
{'1aCw2NNJSZblDbhygVv77kPsL3djmgV5zJZllSOZ_mRk': 'Delete me'}
```

The delete() method will move your spreadsheet to the *Trash* folder on your Google Drive. You can view the contents of your *Trash* folder at *https://drive.google.com/drive/trash*. Notice that spreadsheets in the *Trash* folder will still appear in the dictionary returned by listSpreadsheets(). To permanently delete your spreadsheet, pass True for the permanent keyword argument:

```
>>> ss.delete(permanent=True)
>>> ezsheets.listSpreadsheets()
{}
```

In general, permanently deleting your spreadsheets with automated scripts is not a good idea, because it's impossible to recover a spreadsheet that a bug in your script accidentally deleted. Even free Google Drive accounts have gigabytes of storage available, so you most likely don't need to worry about freeing up space.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Sheet Objects

A Spreadsheet object will have one or more Sheet objects. The Sheet objects represent the rows and columns of data in each sheet. You can access these sheets using the square brackets operator and an integer index.

The Spreadsheet object's sheets attribute holds a tuple of Sheet objects in the order in which they appear in the spreadsheet. To access the Sheet objects in a spreadsheet, enter the following into the interactive shell:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet()  # Starts with a sheet named Sheet1
>>> sheet2 = ss.Sheet('Spam')
>>> sheet3 = ss.Sheet('Eggs')
>>> ss.sheets  # The Sheet objects in this Spreadsheet, in order
(<Sheet sheetId=0, title='Sheet1', rowCount=1000, columnCount=26>, <Sheet sheetId=284204004,
title='Spam', rowCount=1000, columnCount=26>, <Sheet sheetId=1920032872, title='Eggs',
rowCount=1000, columnCount=26>)
>>> ss.sheets[0]  # Gets the first Sheet object in this Spreadsheet
<Sheet sheetId=0, title='Sheet1', rowCount=1000, columnCount=26>
```

The Spreadsheet object's sheetTitles attribute holds a tuple of all the sheet titles. For example, enter the following into the interactive shell:

```
>>> ss.sheetTitles  # The titles of all the Sheet objects in this Spreadsheet
('Sheet1', 'Spam', 'Eggs')
```

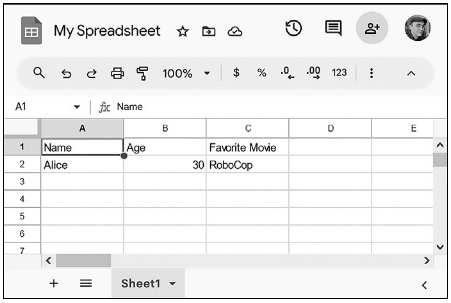
Once you have a Sheet object, you can read data from it and write data to it using the Sheet object's methods, as explained in the next section.

Reading and Writing Data

Just as in Excel, Google Sheets worksheets have columns and rows of cells containing data. You can use the square brackets operator [] to read and write data from and to these cells. For example, to create a new spreadsheet and add data to it, enter the following into the interactive shell:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet()
>>> ss.title = 'My Spreadsheet'
>>> sheet = ss.sheets[0]  # Get the first sheet in this spreadsheet.
>>> sheet.title
'Sheet1'
>>> sheet['A1'] = 'Name'  # Set the value in cell A1.
>>> sheet['B1'] = 'Age'
>>> sheet['C1'] = 'Favorite Movie'
>>> sheet['A1']  # Read the value in cell A1.
'Name'
>>> sheet['A2']  # Empty cells return a blank string.
''
>>> sheet[2, 1]  # Column 2, Row 1 is the same address as B1.
'Age'
>>> sheet['A2'] = 'Alice'
>>> sheet['B2'] = 30
>>> sheet['C2'] = 'RoboCop'
>>> sheet['B2']  # Note that all data is returned as strings.
'30'
```

These instructions should produce a Google Sheets spreadsheet that looks like Figure 15-2.



	A	B	C	D	E
1	Name	Age	Favorite Movie		
2	Alice	30	RoboCop		
3					
4					
5					
6					
7					

Figure 15-2 The spreadsheet created with the example instructions

All of the data in the Sheet object is loaded when the Spreadsheet object is first loaded, so the data is read instantly. However, writing values to the online spreadsheet requires a network connection and can take about a second. If you have thousands of cells to update, updating them one at a time might be quite slow. Instead, the next couple of sections will show you how to update entire rows and columns at once.

Addressing Columns and Rows

Cell addressing works in Google Sheets just like in Excel. The only difference is that, unlike Python's 0-based list indexes, Google Sheets have 1-based columns and rows: the first column or row is at index 1, not 0. You can convert from the 'A2' string-style address to the (column, row) tuple-style address (and vice versa) with the convertAddress() function. The getColumnLetterOf() and getColumnNumberOf() functions will also convert a column address between letters and numbers. For example, enter the following into the interactive shell:

```
>>> import ezsheets
>>> ezsheets.convertAddress('A2')  # Converts addresses...
(1, 2)
>>> ezsheets.convertAddress(1, 2)  # ...and converts them back, too.
'A2'
>>> ezsheets.getColumnLetterOf(2)
'B'
>>> ezsheets.getColumnNumberOf('B')
2
```

```
>>> ezsheets.getColumnLetterOf(999)
'ALK'
>>> ezsheets.getColumnNumberOf('ZZZ')
18278
```

The `'A2'` string-style addresses are convenient if you're typing addresses into your source code. But the `(column, row)` tuple-style addresses are convenient if you're looping over a range of addresses and need a numeric identifier for the column. The `convertAddress()`, `getColumnLetterOf()`, and `getColumnNumberOf()` functions are helpful when you need to convert between the two formats.

Reading and Writing Entire Columns and Rows

As mentioned, writing data one cell at a time can often take too long. Fortunately, EZSheets has `Sheet` methods for reading and writing entire columns and rows at the same time. The `getColumn()`, `getRow()`, `updateColumn()`, and `updateRow()` methods will, respectively, read and write columns and rows. These methods make requests to the Google Sheets servers to update the spreadsheet, so they require that you be connected to the internet. In this section's example, we'll upload *produceSales3.xlsx* from Chapter 14 to Google Sheets. You can download it from this book's online resources. The first eight rows look like Table 15-1.

Table 15-1: The First Eight Rows of the *produceSales3.xlsx* Spreadsheet

	A	B	C	D
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL
2	Potatoes	0.86	21.6	18.58
3	Okra	2.26	38.6	87.24
4	Fava beans	2.69	32.8	88.23
5	Watermelon	0.66	27.3	18.02
6	Garlic	1.19	4.9	5.83
7	Parsnips	2.27	1.1	2.5
8	Asparagus	2.49	37.9	94.37

To upload this spreadsheet, put the *produceSales3.xlsx* file in the current working directory and enter the following into the interactive shell:

```
>>> import ezsheets
>>> ss = ezsheets.upload('produceSales3.xlsx')
>>> sheet = ss.sheets[0]
>>> sheet.getRow(1)  # The first row is row 1, not row 0.
['PRODUCE', 'COST PER POUND', 'POUNDS SOLD', 'TOTAL', '', '']
>>> sheet.getRow(2)
['Potatoes', '0.86', '21.6', '18.58', '', '']
```

```
>>> sheet.getColumn(1)
['PRODUCE', 'Potatoes', 'Okra', 'Fava beans', 'Watermelon', 'Garlic',
--snip--
>>> sheet.getColumn('A')  # The same result as getColumn(1)
['PRODUCE', 'Potatoes', 'Okra', 'Fava beans', 'Watermelon', 'Garlic',
--snip--
>>> sheet.getRow(3)
['Okra', '2.26', '38.6', '87.24', '', '']
>>> sheet.updateRow(3, ['Pumpkin', '11.50', '20', '230'])
>>> sheet.getRow(3)
['Pumpkin', '11.50', '20', '230', '', '']
>>> columnOne = sheet.getColumn(1)
>>> for i, value in enumerate(columnOne):
...     # Make the Python list contain uppercase strings:
...     columnOne[i] = value.upper()
...
>>> sheet.updateColumn(1, columnOne)  # Update the entire column in one request.
```

The getRow() and getColumn() functions retrieve the data from every cell in a specific row or column as a list of values. Note that empty cells become blank string values in the list. You can pass getColumn() either a column number or a letter to tell it to retrieve a specific column's data. The previous example shows that getColumn(1) and getColumn('A') return the same list.

The updateRow() and updateColumn() functions will overwrite the data in the row or column, respectively, with the list of values passed to the function. In this example, the third row initially contains information about okra, but the updateRow() call replaces it with data about pumpkins. Call sheet.getRow(3) again to view the new values in the third row.

Updating cells one at a time is slow if you have many cells to update. Getting a column or row as a list, updating the list, and then updating the entire column or row with the list is much faster, because you can make all changes in one request to Google's cloud services.

To get all of the rows at once, call the getRows() method to return a list of lists. The inner lists inside the outer list each represent a single row of the sheet. You can modify the values in this data structure to change the produce name, pounds sold, and total cost of some of the rows. Then, you can pass it to the updateRows() method by entering the following into the interactive shell:

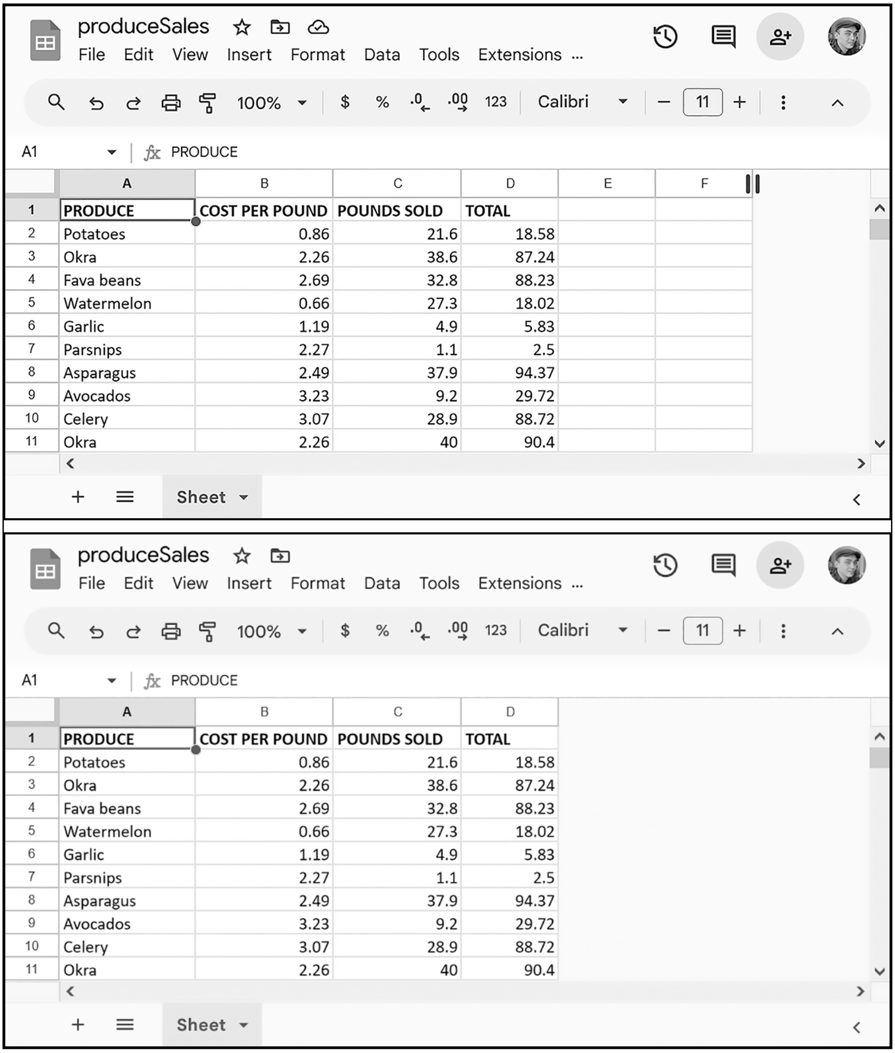
```
>>> rows = sheet.getRows()  # Get every row in the spreadsheet.
>>> rows[0]  # Examine the values in the first row.
['PRODUCE', 'COST PER POUND', 'POUNDS SOLD', 'TOTAL', '', '']
>>> rows[1]
['POTATOES', '0.86', '21.6', '18.58', '', '']
>>> rows[1][0] = 'PUMPKIN'  # Change the produce name.
>>> rows[1]
['PUMPKIN', '0.86', '21.6', '18.58', '', '']
>>> rows[10]
['OKRA', '2.26', '40', '90.4', '', '']
>>> rows[10][2] = '400'  # Change the pounds sold.
>>> rows[10][3] = '904'  # Change the total.
>>> rows[10]
['OKRA', '2.26', '400', '904', '', '']
>>> sheet.updateRows(rows)  # Update the online spreadsheet with the changes.
```

You can update the entire sheet in a single request by passing `updateRows()` the list of lists returned from `getRows()`, amended with the changes made to rows 1 and 10.

Note that the rows in the Google Sheets spreadsheet have empty strings at the end. This is because the uploaded sheet has a column count of `6`, but we have only four columns of data. You can read the number of rows and columns in a sheet with the `rowCount` and `columnCount` attributes. Then, by setting these values, you can change the size of the sheet:

```
>>> sheet.rowCount  # The number of rows in the sheet
23758
>>> sheet.columnCount  # The number of columns in the sheet
6
>>> sheet.columnCount = 4  # Change the number of columns to 4.
>>> sheet.columnCount  # Now the number of columns in the sheet is 4.
4
```

These instructions should delete the fifth and sixth columns of the *produceSales3.xlsx* spreadsheet, as shown in Figure 15-3.



	A	B	C	D	E	F
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL		
2	Potatoes	0.86	21.6	18.58		
3	Okra	2.26	38.6	87.24		
4	Fava beans	2.69	32.8	88.23		
5	Watermelon	0.66	27.3	18.02		
6	Garlic	1.19	4.9	5.83		
7	Parsnips	2.27	1.1	2.5		
8	Asparagus	2.49	37.9	94.37		
9	Avocados	3.23	9.2	29.72		
10	Celery	3.07	28.9	88.72		
11	Okra	2.26	40	90.4		

	A	B	C	D
1	PRODUCE	COST PER POUND	POUNDS SOLD	TOTAL
2	Potatoes	0.86	21.6	18.58
3	Okra	2.26	38.6	87.24
4	Fava beans	2.69	32.8	88.23
5	Watermelon	0.66	27.3	18.02
6	Garlic	1.19	4.9	5.83
7	Parsnips	2.27	1.1	2.5
8	Asparagus	2.49	37.9	94.37
9	Avocados	3.23	9.2	29.72
10	Celery	3.07	28.9	88.72
11	Okra	2.26	40	90.4

Figure 15-3: The sheet before (left) and after (right) changing the column count to four

According to Google's documentation, Google Sheets spreadsheets can have up to 10 million cells in them. However, it's a good idea to make sheets only as big as you need to minimize the time it takes to update and refresh the data.

Creating, Moving, and Deleting Sheets

All Google Sheets spreadsheets start with a single sheet named *Sheet1*. You can add additional sheets to the end of the list of sheets with the `Sheet()` method, which accepts an optional string to use as the new sheet's title. An optional second argument can specify the integer index of the new sheet.

To create a spreadsheet and then add new sheets to it, enter the following into the interactive shell:

```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet()
>>> ss.title = 'Multiple Sheets'
>>> ss.sheetTitles
('Sheet1',)
>>> ss.Sheet('Spam')  # Create a new sheet at the end of the list of sheets.
<Sheet sheetId=2032744541, title='Spam', rowCount=1000, columnCount=26>
>>> ss.Sheet('Eggs')  # Create another new sheet.
<Sheet sheetId=417452987, title='Eggs', rowCount=1000, columnCount=26>
>>> ss.sheetTitles
('Sheet1', 'Spam', 'Eggs')
>>> ss.Sheet('Bacon', 0)  # Create a sheet at index 0 in the list of sheets.
<Sheet sheetId=814694991, title='Bacon', rowCount=1000, columnCount=26>
>>> ss.sheetTitles
('Bacon', 'Sheet1', 'Spam', 'Eggs')
```

These instructions add three new sheets to the spreadsheet: *Bacon*, *Spam*, and *Eggs* (in addition to the default *Sheet1*). The sheets in a spreadsheet are ordered, and new sheets go to the end of the list unless you pass a second argument to `Sheet()` specifying the sheet's index. Here, you create the sheet titled *Bacon* at index `0`, making *Bacon* the first sheet in the spreadsheet and displacing the other three sheets by one position. This is similar to the behavior of the `insert()` list method.

You can see the new sheets on the tabs at the bottom of the screen, as shown in Figure 15-4.

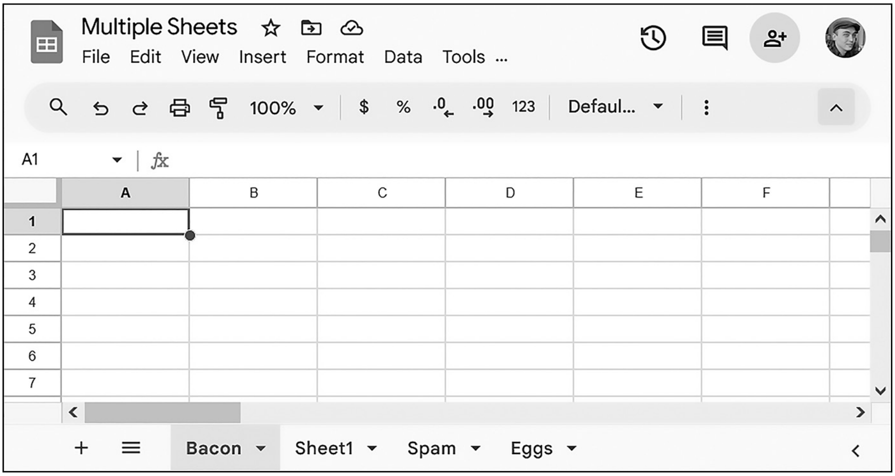


Figure 15-4: The Multiple Sheets *spreadsheet after adding sheets* Spam, Eggs, *and* Bacon

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You can get the order of a sheet from its index attribute and then assign a new index to this attribute to reorder the sheet:

```
>>> ss.sheetTitles
('Bacon', 'Sheet1', 'Spam', 'Eggs')
>>> ss.sheets[0].index
0
>>> ss.sheets[0].index = 2  # Move the sheet at index 0 to index 2.
>>> ss.sheetTitles
('Sheet1', 'Spam', 'Bacon', 'Eggs')
>>> ss.sheets[2].index = 0  # Move the sheet at index 2 to index 0.
>>> ss.sheetTitles
('Bacon', 'Sheet1', 'Spam', 'Eggs')
```

The Sheet object's delete() method will delete the sheet from the spreadsheet. If you want to keep the sheet but delete the data it contains, call the clear() method to clear all the cells and make it a blank sheet. Enter the following into the interactive shell:

```
>>> ss.sheetTitles
('Bacon', 'Sheet1', 'Spam', 'Eggs')
>>> ss.sheets[0].delete()  # Delete the sheet at index 0: the "Bacon" sheet.
>>> ss.sheetTitles
('Sheet1', 'Spam', 'Eggs')
>>> ss['Spam'].delete()  # Delete the "Spam" sheet.
>>> ss.sheetTitles
('Sheet1', 'Eggs')
>>> sheet = ss['Eggs']  # Assign a variable to the "Eggs" sheet.
>>> sheet.delete()  # Delete the "Eggs" sheet.
>>> ss.sheetTitles
('Sheet1',)
>>> ss.sheets[0].clear()  # Clear all the cells on the "Sheet1" sheet.
>>> ss.sheetTitles  # The "Sheet1" sheet is empty but still exists.
('Sheet1',)
```

Deleting sheets is permanent; there's no way to recover the data. However, you can back up sheets by copying them to another spreadsheet with the copyTo() method, as explained in the next section.

Copying Sheets

Every Spreadsheet object has an ordered list of the Sheet objects it contains, and you can use this list to reorder the sheets (as shown in the previous section) or copy them to other spreadsheets. To copy a Sheet object to another Spreadsheet object, call the copyTo() method. Pass it the destination Spreadsheet object as an argument. To create two spreadsheets and copy the first spreadsheet's data to the other sheet, enter the following into the interactive shell:

```
>>> import ezsheets
>>> ss1 = ezsheets.Spreadsheet()
>>> ss1.title = 'First Spreadsheet'
>>> ss1.sheets[0].title = 'Spam'  # ss1 will have a sheet named Spam.
>>> ss2 = ezsheets.Spreadsheet()
>>> ss2.title = 'Second Spreadsheet'
>>> ss2.sheets[0].title = 'Eggs'  # ss2 will have a sheet named Eggs.
>>> ss1[0]
<Sheet sheetId=0, title='Spam', rowCount=1000, columnCount=26>
>>> ss1[0].updateRow(1, ['Some', 'data', 'in', 'the', 'first', 'row'])
>>> ss1[0].copyTo(ss2)  # Copy the ss1's Sheet1 to the ss2 spreadsheet.
>>> ss2.sheetTitles  # ss2 now contains a copy of ss1's Sheet1.
('Eggs', 'Copy of Spam')
```

Copied sheets appear with a prefix of `Copy of` at the end of the list of the destination spreadsheet's sheets. If you wish, you can change their `index` attribute to reorder them in the new spreadsheet.

Google Forms

Your Google account also gives you access to Google Forms at *https://forms.google.com/*. You can create surveys, event registrations, or feedback forms with Google Forms, then receive the answers that users submit in a Google Sheets spreadsheet. Using EZSheets, your Python programs can access this data from the spreadsheet.

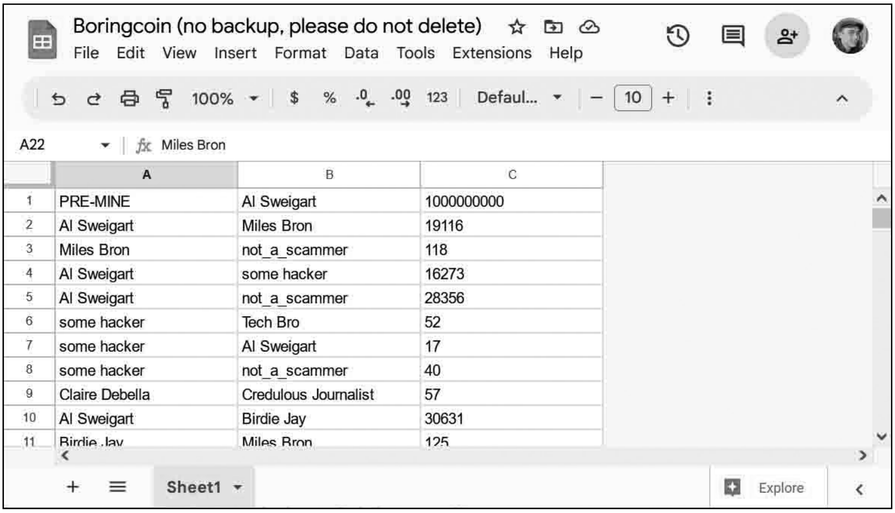
In Chapter 19, you'll learn to schedule your Python programs to run at regular, specified times. You could write a program that regularly checks a Google Forms spreadsheet for responses and detect any new entries it hasn't seen before. Then, using the information in Chapter 20, you can have the program send you a text so that you can get real-time notifications when a form is filled out.

As you've seen, Python is well-known as a "glue" language for tying together multiple existing software systems, letting you create an automated process more powerful than the sum of its parts.

Project 11: Fake Blockchain Cryptocurrency Scam

In this project, we'll use Google Sheets as a fake blockchain to track the transactions of Boringcoin, a cryptocurrency scam I'm promoting. (It turns out that investors and customers don't care if your blockchain product uses a real blockchain data structure; they will give you money anyway.)

The URL *https://autbor.com/boringcoin* redirects to the Google Sheets URL for Boringcoin's blockchain. The spreadsheet has three columns: the sender of the transaction, the recipient of the transaction, and the amount of the transaction. The amount is deducted from the sender and added to the recipient. If the sender is `'PRE-MINE'`, this money is created out of thin air and added to the recipient account. Figure 15-5 shows this Google Sheet.



	A	B	C
1	PRE-MINE	Al Sweigart	1000000000
2	Al Sweigart	Miles Bron	19116
3	Miles Bron	not_a_scammer	118
4	Al Sweigart	some hacker	16273
5	Al Sweigart	not_a_scammer	28356
6	some hacker	Tech Bro	52
7	some hacker	Al Sweigart	17
8	some hacker	not_a_scammer	40
9	Claire Debella	Credulous Journalist	57
10	Al Sweigart	Birdie Jay	30631
11	Birdie Jay	Miles Bron	125

Figure 15-5: The fake blockchain for Boringcoin, stored on a Google Sheet

The first transaction has the sender `'PRE-MINE'` and the recipient `'Al Sweigart'`, and the amount is a humble `1000000000`. The `'Al Sweigart'` account then transfers `19116` Boringcoins to `'Miles Bron'`, who then transfers `118` Boringcoins to `'not_a_scammer'`. The fourth transaction transfers `16273` Boringcoins from `'Al Sweigart'` to `'some hacker'`. (I did not authorize this transaction and have since stopped using *python12345* as my Google account password.)

Let's write two programs. First, the *auditBoringcoin.py* program examines all the transactions and generates a dictionary of all accounts and their current balance. Second, the *addBoringcoinTransaction.py* program adds a row to the end of the Google Sheets for a new transaction. These blockchain programs are just for fun and not real (though "real" blockchain projects such as NFTs and "web3" are just as much a fantasy).

Step 1: Audit the Fake Blockchain

We need to write a program to examine the entire "blockchain" and determine the current balance of all accounts. We'll use a dictionary to hold this data, where the keys are strings of the account name and the values are integers of how many Boringcoins are in them. We also want the program to display how many total Boringcoins are in the cryptocurrency network. We can start by importing EZSheets and setting up the dictionary:

```
import ezsheets
ss = ezsheets.Spreadsheet('https://autbor.com/boringcoin')
accounts = {}  # Keys are names, and values are amounts.
```

Next, we'll loop through every row in the spreadsheet, identifying the sender, recipient, and amount. Keep in mind that Google Sheets always

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

returns data as a string, so we need to convert it to an integer to do math with the amount value:

```
# Each row is a transaction. Loop over each one:
for row in ss.sheets[0].getRows():
    sender, recipient, amount = row[0], row[1], int(row[2])
```

If the sender is the special account 'PRE-MINE', then it is simply a source of infinite money into other accounts. All of the best cryptocurrency scams use pre-mined coins, and ours is no exception. Add the amount to the recipient account in the accounts dictionary. The setdefault() method sets the value of the account to 0 if it doesn't already exist in the dictionary:

```
    if sender == 'PRE-MINE':
        # The 'PRE-MINE' sender invents money out of thin air.
        accounts.setdefault(recipient, 0)
        accounts[recipient] += amount
```

Otherwise, we should deduct the amount from the sender and add it to the recipient:

```
    else:
        # Move funds from the sender to the recipient.
        accounts.setdefault(sender, 0)
        accounts.setdefault(recipient, 0)
        accounts[sender] -= amount
        accounts[recipient] += amount
```

After the loop finishes, we can see the current balances by printing the accounts dictionary.

```
print(accounts)
```

As part of our audit, let's also go through this dictionary and add up the totals of everyone's balance to find out how many Boringcoins are in the entire network. Start a total variable at 0, and then have a for loop go through each value in the key-value pairs of the accounts dictionary. After adding each value to total, we can print the total amount of Boringcoins:

```
total = 0
for amount in accounts.values():
    total += amount
print('Total Boringcoins:', total)
```

When we run this program, the output looks like this:

```
{'Al Sweigart': 999058553, 'Miles Bron': 38283, 'not_a_scammer': 48441,
'some hacker': 44429, 'Tech Bro': 53424, 'Claire Debella': 54443,
'Credulous Journalist': 50408, 'Birdie Jay': 36832, 'Carol': 82867, 'Mark Z.':
 68650, 'Bob': 37920, 'Andi Brand': 57218, 'Eve': 88296, 'Al Sweigart sock
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
#27': 78080, 'Tax evader': 40937, 'Duke Cody': 17544, 'Lionel Toussaint':
54650, 'some scammer': 2694, 'Alice': 44503, 'David': 41828}
Total Boringcoins: 1000000000
```

The total is 1000000000, which makes sense, because that's how many Boringcoins were pre-mined.

Step 2: Make Transactions

The next program, *addBoringcoinTransaction.py*, adds additional rows to the "blockchain" Google Sheet to add new transactions. It reads three command line arguments from the list in sys.argv: the sender, the recipient, and the amount. For example, you could run the following from the terminal:

```
python addBoringcoinTransaction.py "Al Sweigart" Eve 2000
```

The program would access the Google Sheet, add a blank row to the bottom, and then fill it in with the values 'Al Sweigart', 'Eve', and '2000'. Note that in the terminal, you'll need to enclose any command line argument that contains a space in double quotes, like "Al Sweigart"; otherwise, the terminal will think they are two separate arguments.

The start of *addBoringcoinTransactions.py* checks the command line arguments and assigns the sender, recipient, and amount variables based on them:

```
import sys, ezsheets

if len(sys.argv) < 4:
    print('Usage: python addBoringcoinTransaction.py sender recipient amount')
    sys.exit()

# Get the transaction info from the command line arguments:
sender, recipient, amount = sys.argv[1:]
```

You won't need to convert amount from a string to an integer, because we'll be writing it as a string to the spreadsheet.

Next, EZSheets connects to the Google Sheets containing the fake blockchain and selects the first sheet in the spreadsheet (at index 0). Note that you don't have permission to edit the Boringcoin Google Sheets, so open that URL in a web browser while logged in to your Google account and then select **File ▸ Make a Copy** to copy it to your Google Account. Then, replace the 'https://autbor.com/boringcoin' string with a string of your Google Sheet's URL from the browser address bar:

```
# Change this URL to your copy of the Google Sheet, or else you'll
# get a "The caller does not have permission" error.
ss = ezsheets.Spreadsheet('https://autbor.com/boringcoin')
sheet = ss.sheets[0]
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Finally, you should get the number of rows in the sheet, increase it by one, and then fill in the columns of this row with the sender, recipient, and amount data:

```
# Add one more row to the sheet for a new transaction:
sheet.rowCount += 1

sheet[1, sheet.rowCount] = sender
sheet[2, sheet.rowCount] = recipient
sheet[3, sheet.rowCount] = amount
```

Now when you run `python addBoringcoinTransaction.py "Al Sweigart" Eve 2000` from the terminal, the Google Sheets will have a new row with *Al Sweigart*, *Eve*, and *2000* added at the bottom. You can rerun the *auditBoringcoin.py* program to see the updated account balances of everyone in the cryptocurrency network.

The use of Google Sheets for our blockchain data structure is irresponsible, error prone, and a security catastrophe waiting to happen. This makes it on par with most marketed blockchain products. Don't miss out! Contact me to get in on this limited offer to buy Boringcoin before the pyramid scheme collapses!

Working with Google Sheets Quotas

Because Google Sheets is online, you can easily share sheets among multiple users who can all access the sheets simultaneously. However, this also means that reading and updating the sheets will be slower than reading and updating Excel files stored locally on your hard drive. In addition, Google Sheets limits how many read and write operations you can perform.

According to Google's developer guidelines, users are restricted to creating 250 new spreadsheets a day, and free Google accounts can perform a few hundred requests per minute. You can find Google's usage limits at *https://developers.google.com/sheets/api/limits*. Attempting to exceed this quota will raise the `googleapiclient.errors.HttpError` "Quota exceeded for quota group" exception. EZSheets will automatically catch this exception and retry the request. When this happens, the function calls to read or write data will take several seconds (or even a full minute or two) before they return. If the request continues to fail (which is possible if another script using the same credentials is also making requests), EZSheets will re-raise this exception.

This means that, on occasion, your EZSheets method calls may take several seconds before they return. If you want to view your API usage or increase your quota, go to the IAM & Admin Quotas page at *https://console.developers.google.com/iam-admin/quotas* to learn about paying for increased usage. If you'd rather just deal with the `HttpError` exceptions yourself, you can set `ezsheets.IGNORE_QUOTA` to `True`, and EZSheets' methods will raise these exceptions when it encounters them.

Summary

Google Sheets is a popular online spreadsheet application that runs in your browser. Using the EZSheets third-party package, you can download, create, read, and modify spreadsheets. EZSheets represents spreadsheets as `Spreadsheet` objects, each of which contains an ordered list of `Sheet` objects. Each sheet has columns and rows of data that you can read and update in several ways.

While Google Sheets makes sharing data and cooperative editing easy, its main disadvantage is speed: you must update spreadsheets with web requests, which can take a few seconds to execute. But for most purposes, this speed restriction won't affect Python scripts using EZSheets. Google Sheets also limits how often you can make changes.

For complete documentation of EZSheets' features, visit *https://ezsheets.readthedocs.io/*.

Practice Questions

1. What three files do you need for EZSheets to access Google Sheets?
2. What two types of objects does EZSheets have?
3. How can you create an Excel file from a Google Sheets spreadsheet?
4. How can you create a Google Sheets spreadsheet from an Excel file?
5. The `ss` variable contains a `Spreadsheet` object. What code will read data from the cell B2 in a sheet titled *Students*?
6. How can you find the column letters for column 999?
7. How can you find out how many rows and columns a sheet has?
8. How do you delete a spreadsheet? Is this deletion permanent?
9. What functions will create a new `Spreadsheet` object and a new `Sheet` object, respectively?
10. What would happen if, by making frequent read and write requests with EZSheets, you exceed your Google account's quota?

Practice Programs

For practice, write programs to do the following tasks.

Downloading Google Forms Data

I mentioned earlier that Google Forms allows you to create simple online forms that make it easy to collect information from people. The information entered into a form is stored in a Google Sheets spreadsheet. For this project, write a program that can automatically download the form information that users have submitted. Go to *https://docs.google.com/forms/* and start a new blank form. Add fields to the form that ask the user for a name

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

and email address. Then, click the **Send** button in the upper right to get a link to your new form. Try to enter a few example responses into this form.

On the *Responses* tab of your form, click the green **Create Spreadsheet** button to create a Google Sheets spreadsheet that will hold the responses that users submit. You should see your example responses in the first rows of this spreadsheet. Then, write a Python script using EZSheets to collect a list of the email addresses on this spreadsheet.

Converting Spreadsheets to Other Formats

You can use Google Sheets to convert a spreadsheet file to other formats. Write a script that passes a submitted file to `upload()`. Once the spreadsheet has uploaded to Google Sheets, download it using `downloadAsExcel()`, `downloadAsODS()`, and other such functions to create a copy of the spreadsheet in these other formats.

Finding Mistakes in a Spreadsheet

After a long day at the bean-counting office, I've finished a spreadsheet with all the bean totals and uploaded them to Google Sheets. The spreadsheet is publicly viewable (but not editable). You can get this spreadsheet with the following code:

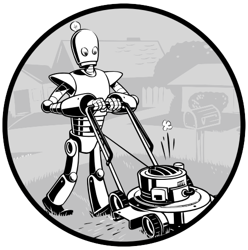
```
>>> import ezsheets
>>> ss = ezsheets.Spreadsheet('1jDZEdvSIh4TmZxccyy0ZXrH-ELlrwq8_YYiZrEOB4jg')
```

View the spreadsheet in your browser by going to *https://docs.google.com/spreadsheets/d/1jDZEdvSIh4TmZxccyy0ZXrH-ELlrwq8_YYiZrEOB4jg*. The columns of the first sheet in this spreadsheet are *BEANS PER JAR*, *JARS*, and *TOTAL BEANS*. The *TOTAL BEANS* column is the product of the numbers in the *BEANS PER JAR* and *JARS* columns. However, there is a mistake in one of the 15,000 rows in this sheet. That's too many rows to check by hand. Luckily, you can write a script that checks the totals.

As a hint, you can access the individual cells in a row with `ss.sheets[0].getRow(rowNum)`, where `ss` is the `Spreadsheet` object and *`rowNum`* is the row number. Remember that row numbers in Google Sheets begin at 1, not 0. The cell values will be strings, so you'll need to convert them to integers before your program can work with them. The expression `int(ss.sheets[0].getRow(2)[0]) * int(ss.sheets[0].getRow(2)[1]) == int(ss.sheets[0].getRow(2)[2])` evaluates to `True` if row 2 has the correct total. Put this code in a loop to identify which row in the sheet has the incorrect total.

16

SQLITE DATABASES



You're probably used to organizing information into spreadsheets such as Excel or Google Sheets, but most software stores its data in applications called *databases*. Databases make it easy for your programs to retrieve the specific data you want. If you had a spreadsheet or text file of cats and wanted to find the fur color of a cat named Zophie, you could press CTRL-F and enter "Zophie." But what if you wanted to find the fur color of all cats that weighed between 3 and 5 kilograms and were born before October 2023? Even with the regular expressions from Chapter 9, this would be a tricky thing to code.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Databases allow you to perform complex queries like this one, written in the mini language of *Structured Query Language (SQL)*. You'll see the term *SQL* used to refer to both a language for database operations and the databases that understand this language; it's often pronounced "es-cue-el" but also sometimes "sequel." This chapter introduces you to SQL and database concepts using *SQLite* (pronounced either "sequel-ite," "es-cue-lite," or "es-cue-el-ite"), a lightweight database included with Python.

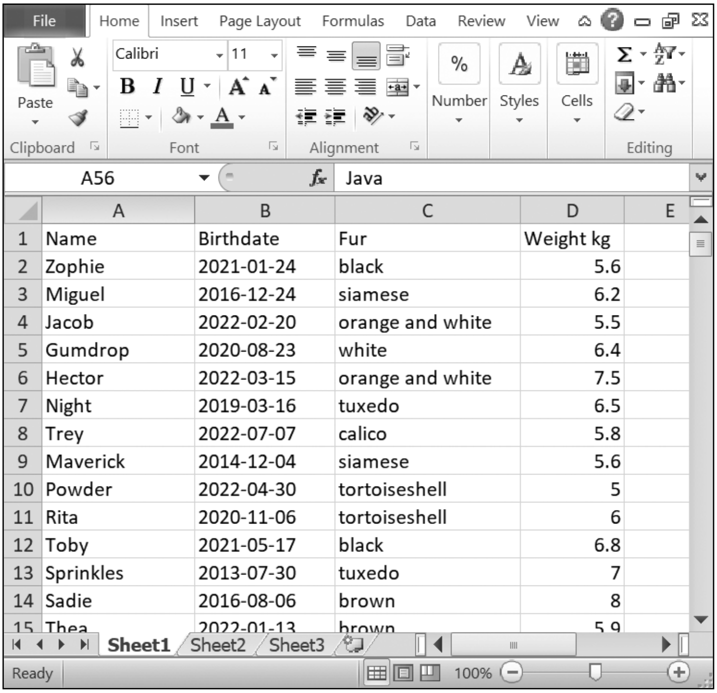
SQLite is the most widely deployed database software, as it runs on every operating system and is small enough to embed within other applications. At the same time, SQLite's simplifications make it notably different from other databases. While large database software such as PostgreSQL, MySQL, Microsoft SQL Server, and Oracle are intended to run on dedicated server hardware accessed over a network, SQLite stores the entire database in a single file on your computer.

Even if you're already familiar with SQL databases, SQLite has enough of its own quirks that you should read this chapter to learn how to make the most of it. You can find the online SQLite documentation at *https://sqlite.org/docs.html* and the Python `sqlite3` module documentation at *https://docs.python.org/3/library/sqlite3.html*.

Spreadsheets vs. Databases

Let's consider the similarities and differences between spreadsheets and databases. In a spreadsheet, rows contain individual records, while columns represent the kind of data stored in the fields of each record. For example, Figure 16-1 is a spreadsheet of some of my cats. The columns list the name, birthday, fur color, and weight (in kilograms) of each cat.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart



A56 | Java

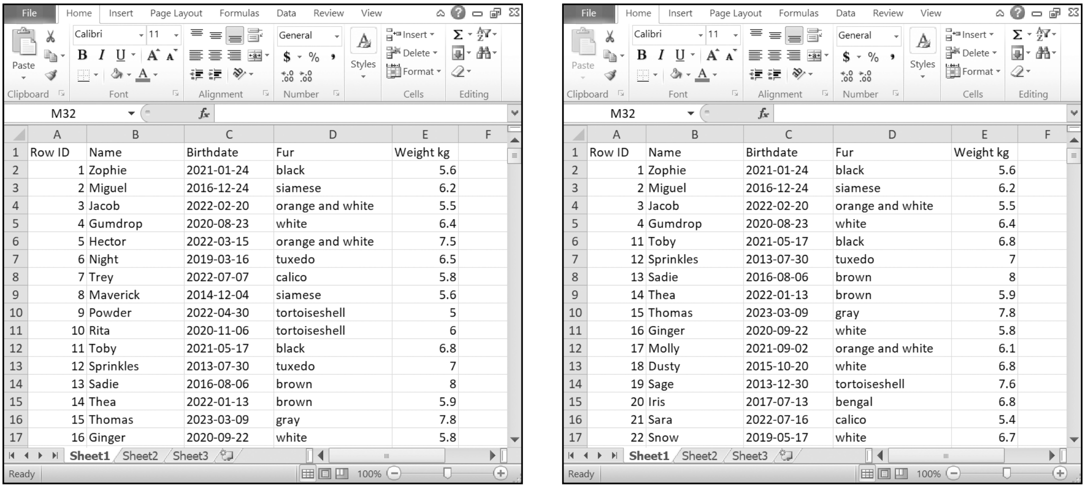
	A	B	C	D	E
1	Name	Birthdate	Fur	Weight kg	
2	Zophie	2021-01-24	black	5.6	
3	Miguel	2016-12-24	siamese	6.2	
4	Jacob	2022-02-20	orange and white	5.5	
5	Gumdrop	2020-08-23	white	6.4	
6	Hector	2022-03-15	orange and white	7.5	
7	Night	2019-03-16	tuxedo	6.5	
8	Trey	2022-07-07	calico	5.8	
9	Maverick	2014-12-04	siamese	5.6	
10	Powder	2022-04-30	tortoiseshell	5	
11	Rita	2020-11-06	tortoiseshell	6	
12	Toby	2021-05-17	black	6.8	
13	Sprinkles	2013-07-30	tuxedo	7	
14	Sadie	2016-08-06	brown	8	
15	Thea	2022-01-13	brown	5.9	

Figure 16-1: A spreadsheet stores data records as rows with a set column structure.

We can store this same information in a database. You can think of a database *table* as a spreadsheet, and a database can contain one or more tables. Tables have columns of different properties for each *record*, also called a *row* or *entry*. Databases like SQLite are called *relational databases*, where *relation* means that the database can contain multiple tables with relationships between them, as you'll later see.

Both spreadsheets and databases label the data they contain. A spreadsheet automatically labels the columns with letters and the rows with numbers. In addition, the example cat spreadsheet uses its first row to give the columns descriptive names. Each of the subsequent rows represents exactly one cat. In a SQL database, tables often have an ID column for each record's *primary key*: a unique integer that can unambiguously identify the record. In SQLite, this column is called `rowid`, and SQLite automatically adds it to your tables.

Deleting a spreadsheet row moves up all the rows underneath it, changing their row numbers. But a database record's primary key ID is unique and doesn't change. This is useful in many situations. What if a cat were renamed or had a change in weight? What if we wanted to reorder the rows to list the cats alphabetically by name? Each cat needs a unique identification number that remains constant no matter how the data changes. We could add a Row ID column to our spreadsheet to simulate a SQLite table's `rowid` column. This ID value would stay the same even if rows were deleted or moved around the spreadsheet, as shown in Figure 16-2, where the cats with the Row IDs of 5 to 10 are deleted.

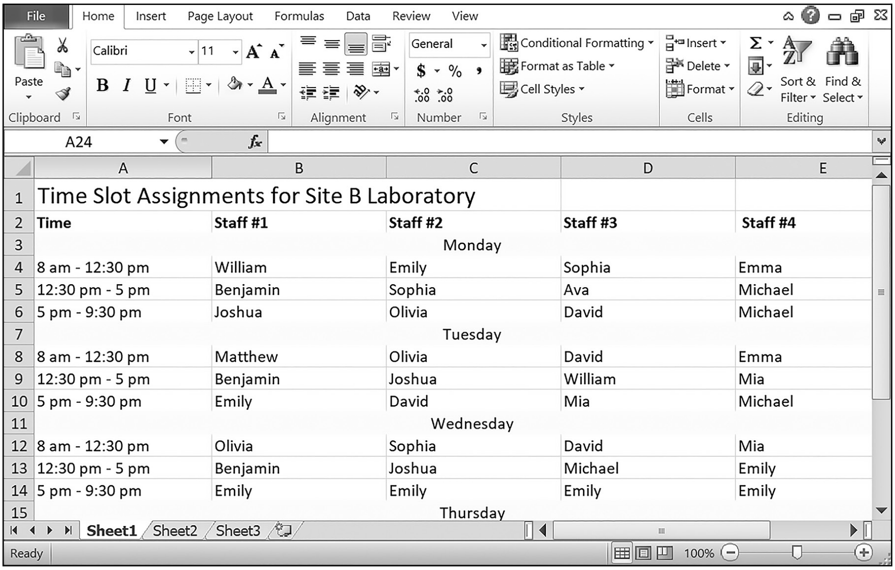


	A	B	C	D	E	F
1	Row ID	Name	Birthdate	Fur	Weight kg	
2	1	Zophie	2021-01-24	black	5.6	
3	2	Miguel	2016-12-24	siamese	6.2	
4	3	Jacob	2022-02-20	orange and white	5.5	
5	4	Gumdrop	2020-08-23	white	6.4	
6	5	Hector	2022-03-15	orange and white	7.5	
7	6	Night	2019-03-16	tuxedo	6.5	
8	7	Trey	2022-07-07	calico	5.8	
9	8	Maverick	2014-12-04	siamese	5.6	
10	9	Powder	2022-04-30	tortoiseshell	5	
11	10	Rita	2020-11-06	tortoiseshell	6	
12	11	Toby	2021-05-17	black	6.8	
13	12	Sprinkles	2013-07-30	tuxedo	7	
14	13	Sadie	2016-08-06	brown	8	
15	14	Thea	2022-01-13	brown	5.9	
16	15	Thomas	2023-03-09	gray	7.8	
17	16	Ginger	2020-09-22	white	5.8	

	A	B	C	D	E	F
1	Row ID	Name	Birthdate	Fur	Weight kg	
2	1	Zophie	2021-01-24	black	5.6	
3	2	Miguel	2016-12-24	siamese	6.2	
4	3	Jacob	2022-02-20	orange and white	5.5	
5	4	Gumdrop	2020-08-23	white	6.4	
6	11	Toby	2021-05-17	black	6.8	
7	12	Sprinkles	2013-07-30	tuxedo	7	
8	13	Sadie	2016-08-06	brown	8	
9	14	Thea	2022-01-13	brown	5.9	
10	15	Thomas	2023-03-09	gray	7.8	
11	16	Ginger	2020-09-22	white	5.8	
12	17	Molly	2021-09-02	orange and white	6.1	
13	18	Dusty	2015-10-20	white	6.8	
14	19	Sage	2013-12-30	tortoiseshell	7.6	
15	20	Iris	2017-07-13	bengal	6.8	
16	21	Sara	2022-07-16	calico	5.4	
17	22	Snow	2019-05-17	white	6.7	

Figure 16-2: The Row ID number, unlike the spreadsheet row numbers, offers a unique identifier for each record (left) even after cats with IDs 5 to 10 are deleted (right).

There is a second way people use spreadsheets that is entirely unlike how databases tend to store data. Spreadsheets can serve as templates for forms rather than as row-based data storage. You may have seen spreadsheets such as Figure 16-3.



	A	B	C	D	E
1	Time Slot Assignments for Site B Laboratory				
2	**Time**	**Staff #1**	**Staff #2**	**Staff #3**	**Staff #4**
3			Monday		
4	8 am - 12:30 pm	William	Emily	Sophia	Emma
5	12:30 pm - 5 pm	Benjamin	Sophia	Ava	Michael
6	5 pm - 9:30 pm	Joshua	Olivia	David	Michael
7			Tuesday		
8	8 am - 12:30 pm	Matthew	Olivia	David	Emma
9	12:30 pm - 5 pm	Benjamin	Joshua	William	Mia
10	5 pm - 9:30 pm	Emily	David	Mia	Michael
11			Wednesday		
12	8 am - 12:30 pm	Olivia	Sophia	David	Mia
13	12:30 pm - 5 pm	Benjamin	Joshua	Michael	Emily
14	5 pm - 9:30 pm	Emily	Emily	Emily	Emily
15			Thursday		

Figure 16-3: A spreadsheet with a lot of formatting and a fixed sized is generally not a good fit for a database.

These spreadsheets tend to have a lot of formatting, with background colors, merged cells, and different fonts, so that they look good to human eyes. While the row-based data spreadsheets can expand infinitely downward as new data is added, these spreadsheets usually have a fixed size and fill-in-the-blank design. They're often meant for humans to print out and look at, rather than for a Python program to extract data from them.

Databases aren't visually pleasing; they just contain raw data. More importantly, while spreadsheets give you the flexibility of putting any data into any cell, databases have a stricter structure to make data retrieval easier for software. If your data tends to look like the example in Figure 16-3, you may be better off storing it in JSON files, or using the `openpyxl` module from Chapter 14 and the EZSheets library from Chapter 15 and leaving it in an Excel or Google spreadsheet.

SQLite vs. Other SQL Databases

If you're used to working with other SQL databases, you might be wondering how SQLite compares. In short, SQLite strikes a balance between simplicity and power. It's a full relational database that uses SQL to read and write massive amounts of data, but it runs within your Python program and operates on a single file. Your program imports the `sqlite3` module just as it would import `sys`, `math`, or any other module in the Python standard library.

Here are the main differences between SQLite and other database software:

- SQLite databases are stored in a single file, which you can move, copy, or back up like any other file.
- SQLite can run on computers with few resources, such as embedded devices or decades-old laptops.
- SQLite is serverless; it doesn't require a background server application to constantly run on your laptop, or any dedicated server hardware. There are no network connections involved.
- From the perspective of users, SQLite doesn't require any installation or configuration. It's part of the Python program.
- For faster performance, SQLite databases can exist entirely in memory and be saved to a file before the program exits.
- While SQLite columns have data types, such as numbers and text, just as other SQL databases do, SQLite doesn't strictly enforce a column's data type.
- There are no permission settings or user roles in SQLite. SQLite has no `GRANT` or `REVOKE` statements like in other SQL databases.
- SQLite is public domain software; you can use it commercially or any way you want without restriction.

The main disadvantage of SQLite is that it can't efficiently handle hundreds or thousands of simultaneous write operations (say, from a social

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

media web app). Aside from that, SQLite is just as powerful as any database, able to reliably handle GBs or even TBs of data, as well as simultaneous read operations, quickly and easily.

SQLite sells itself not so much as a competitor to other database software but as a competitor to using the `open()` function to work with text files (or the JSON, XML, and CSV files you'll learn about in Chapter 18). If your program needs the ability to store and quickly retrieve large amounts of data, SQLite is a better alternative to JSON or spreadsheet files.

Creating Databases and Tables

Let's begin by creating our first database and table using SQL. SQL is a mini language you can work with from within Python, much like regex for regular expressions. Like regex, SQL queries are written as Python string values. And just as you could write your own Python code to perform the text pattern-matching that regular expressions perform, you *could* write your own custom Python code to search for matching data in Python dictionaries and lists. But writing regular expressions and SQL database queries makes these tasks much simpler in the long run, even if they first require you to learn a new skill. Let's explore how to write queries that create tables in a new database.

We'll create a sample SQLite database in a file named *example.db* to store information about cats. To create a database, first import the `sqlite3` module. (The *3* is for SQLite major version 3, which is unrelated to Python 3.) A SQLite database lives in a single file. The name of the file can be anything, but by convention we give it a *.db* file extension. The extension *.sqlite* is also commonly used.

A database can contain multiple tables, and each table should store one particular type of data. For example, one table could contain records of cats, while another table could contain records of vaccinations given to specific cats in the first table. You can think of a table as a list of tuples, where each tuple is a table row. The `cats` table is essentially the same as `[('Zophie', '2021-01-24', 'black', 5.6), ('Colin', '2016-12-24', 'siamese', 6.2), ...]`, and so on.

Let's create a database, then create a table for the cat data, insert some cat records into it, read the data from the database, and close the database connection.

Connecting to Databases

The first step of writing SQLite code is getting a `Connection` object for the database file by calling `sqlite3.connect()`. Enter the following into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
```

The first argument to the function can be either a string of a filename or a pathlib.Path object for the database file. If this filename doesn't belong to an existing SQLite database, the function creates a new file containing an empty database. For example, sqlite3.connect('example.db', isolation_level =None) connects a database file named *example.db* in the current working directory. If this file doesn't exist, the function creates an empty one.

If the file you connect to exists but isn't a SQLite database file, Python raises a sqlite3.DatabaseError: file is not a database exception once you try to execute queries. "Checking Path Validity" in Chapter 10 explains how to use the exists() Path method and os.path.exists() function, which can tell your program if a file exists or not.

The isolation_level=None keyword argument causes the database to use *write-ahead logging*, or WAL mode. The details of WAL mode are beyond the scope of this book, but in general, "always use WAL mode" is good advice for beginners.

The sqlite3.connect() function returns a Connection object, which we store in a variable named conn for these examples. Each Connection object connects to one SQLite database file. You can, of course, choose any variable name you'd like for this Connection object, and you should use more descriptive variable names if your program opens multiple databases at the same time. But for small programs that connect to only one database at a time, the name conn is easy to write and descriptive enough. (The name con would be even shorter, but is easy to misunderstand as "*con*sole," "*con*tent," or "*con*fusing name for a variable.")

When your program is done with the database, call conn.close() to close the connection. The program also closes the connection automatically when it terminates.

Creating Tables

After connecting to a new, blank database, create a table with a CREATE TABLE SQL query. To run SQL queries, you must call the execute() method of Connection objects. Pass this conn.execute() method a string of the query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
>>> conn.execute('CREATE TABLE IF NOT EXISTS cats (name TEXT NOT NULL,
birthdate TEXT, fur TEXT, weight_kg REAL) STRICT')
<sqlite3.Cursor object at 0x00000201B2399540>
```

By convention, SQL keywords, such as CREATE and TABLE, are written using uppercase letters. However, SQLite doesn't enforce this; the query 'create table if not exists cats (name text not null, birthdate text, fur text, weight_kg real) strict' runs just fine. Table and column names are also case-insensitive, but the convention is to make them lowercase and to separate multiple words with underscores, as in weight_kg.

The CREATE TABLE statement raises a sqlite3.OperationalError: table cats already exists exception if you try to create a table that already exists

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

without the IF NOT EXISTS part. Including this part is a quick way to avoid tripping over this exception, and you'll almost always want to add it to your CREATE TABLE queries.

In our example, we follow the CREATE TABLE IF NOT EXISTS keywords with the table name cats. After the table name is a set of parentheses containing the column names and data types.

Defining Data Types

There are six data types in SQLite:

NULL Analogous to Python's None

INT or INTEGER Analogous to Python's int type

REAL A reference to the mathematics term *real number*; analogous to Python's float type

TEXT Analogous to Python's str type

BLOB Short for *Binary Large Object*; analogous to Python's bytes type and useful for storing entire files in a database

ANY For columns that can store data of any type

SQLite has its own data types because it wasn't built just for Python; other programming languages can interact with SQLite databases as well.

Unlike other SQL database software, SQLite isn't strict about the data types of its columns. This means SQLite will, by default, gladly store the string 'Hello' in an INTEGER column without raising an exception. But SQLite's data types aren't entirely cosmetic, either; SQLite automatically *casts* (that is, changes) data to the column's data type if possible, a feature called *type affinity*. For example, if you add the string '42' to an INTEGER column, SQLite automatically stores the value as the integer 42, because the column has a type affinity for integers. However, if you add the string 'Hello' to an INTEGER column, SQLite will store 'Hello' (without error), because despite the integer type affinity, 'Hello' cannot be converted to an integer. If you want to disable this automatic type casting for a column, use the ANY data type to tell SQLite that the column has no type affinity.

The STRICT keyword enables *strict mode* for this table. Under strict mode, every column must be given a data type, and SQLite will raise a sqlite3.IntegrityError exception if you try to insert data of the wrong type into the table. SQLite will still automatically cast data to the column's data type; inserting '42' into an INTEGER column would insert the integer 42. However, the string 'Hello' cannot be cast to an integer, so attempting to insert it would raise an exception. I highly recommend using strict mode; it can give you an early warning about bugs caused by inserting incorrect data into your table. If your table doesn't need to enforce data types, you can use strict mode anyway and set the columns to the ANY data type.

SQLite added the STRICT keyword in version 3.37.0, which is used by Python 3.11 and later. Earlier versions don't know about strict mode and will report a syntax error if you attempt to use it. You can always check the version of SQLite that Python is using by examining the sqlite3.sqlite_version variable, which will look something like this:

```
>>> import sqlite3
>>> sqlite3.sqlite_version
'3.xx.xx'
```

SQLite doesn't have a Boolean data type, so use INTEGER for Boolean data instead; you can store a 1 to represent True and a 0 to represent False. SQLite also doesn't have a date, time, or datetime data type. Instead, you can use the TEXT data type to store a string in a format listed in Table 16-1.

Table 16-1: Recommended Formats for Dates, Times, and Datetimes in SQLite

Format	Example
YYYY-MM-DD	'2035-10-31'
YYYY-MM-DD HH:MM:SS	'2035-10-31 16:30:00'
YYYY-MM-DD HH:MM:SS.SSS	'2035-10-31 16:30:00.407'
HH:MM:SS	'16:30:00'
HH:MM:SS.SSS	'16:30:00.407'

The NOT NULL part of name TEXT NOT NULL specifies that the Python None value cannot be stored in the name column. This is a good way to make a table column mandatory.

SQLite tables automatically create a rowid column containing a unique primary key integer. Even if your cats table has two cats that coincidentally have the same name, birthday, fur color, and weight, the rowid allows you to distinguish between them.

Listing Tables and Columns

All SQLite databases have a table named sqlite_schema that lists metadata about the database, including all of its tables. To list the tables in the SQLite database, run the following query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type="table"').fetchall()
[('cats',)]
```

The output shows the cats table we just created. (I explain the syntax of the SELECT statement in "Reading Data from the Database" on page XX.) To obtain information about the columns in the cats table, run the following query:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> conn.execute('PRAGMA TABLE_INFO(cats)').fetchall()
[(0, 'name', 'TEXT', 1, None, 0), (1, 'birthdate', 'TEXT', 0, None, 0), (2,
'fur', 'TEXT', 0, None, 0), (3, 'weight_kg', 'REAL', 0, None, 0)]
```

This query returns a list of tuples that each describe a column of the table. For example, the `(1, 'birthdate', 'TEXT', 0, None, 0)` tuple provides the following information about the `birthdate` column:

Column position The `1` indicates that the column is second in the table. Column numbers are zero based, like Python list indexes, so the first column is at position 0.

Name `'birthdate'` is the name of the column. Remember that SQLite column and table names are case insensitive.

Data type `'TEXT'` is the SQLite data type of the `birthdate` column.

Whether the column is `NOT NULL` The `0` means `False` and that the column is not `NOT NULL` (that is, you can put `None` values in this column).

Default value `None` is the default value inserted if no other value is specified.

Whether the column is the primary key The `0` means `False`, meaning this column is not a primary-key column.

Note that the `sqlite_schema` table itself isn't listed as a table. You'll never need to modify the `sqlite_schema` table yourself, and doing so will likely corrupt the database, making it unreadable.

CRUD Database Operations

CRUD is an acronym for the four basic operations that databases carry out: creating data, reading data, updating data, and deleting data. In SQLite, we perform these operations with `INSERT`, `SELECT`, `UPDATE`, and `DELETE` statements, respectively. Here are examples of each statement, which we'll later pass as strings to `conn.execute()`:

- `INSERT INTO cats VALUES ("Zophie", "2021-01-24", "black", 5.6)`
- `SELECT rowid, * FROM cats ORDER BY fur`
- `UPDATE cats SET fur = "gray tabby" WHERE rowid = 1`
- `DELETE FROM cats WHERE rowid = 1`

Most applications and social media websites are really just fancy user interfaces for a CRUD database. When you post a photo or reply, you're creating a record in a database somewhere. When you scroll through a social media timeline, you're reading records from the database. And when you edit or delete a post, you're performing an update or a deletion operation. Whenever you're learning a new app, programming language, or query language, use the CRUD acronym to remind yourself of which basic operations you should find out about.

Inserting Data into the Database

Now that we've created the database and a cats table, let's insert records for my pet cats. I have about 300 cats in my home, and using a SQLite database helps me keep track of them. An INSERT statement can add new records to a table. Enter the following code into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
>>> conn.execute('CREATE TABLE IF NOT EXISTS cats (name TEXT NOT NULL, birthdate TEXT,
fur TEXT, weight_kg REAL) STRICT')
<sqlite3.Cursor object at 0x00000201B2399540>
>>> conn.execute('INSERT INTO cats VALUES ("Zophie", "2021-01-24", "black", 5.6)')
<sqlite3.Cursor object at 0x00000162842E78C0>
```

This INSERT query adds a new row to the cats table. Inside the parentheses are the comma-separated values for its columns. The parentheses are mandatory for INSERT queries. Note that when inserting TEXT values, I've used double quotation marks (") because I'm already using single quotation marks (') for the query's string. The sqlite3 module uses either single or double quotes for its TEXT values.

Transactions

An INSERT statement begins a *transaction*, which is a unit of work in a database. Transactions must pass the *ACID test*, a database concept meaning that transactions are:

Atomic The transaction is carried out either completely or not at all.

Consistent The transaction doesn't violate constraints, such as NOT NULL rules for columns.

Isolated One transaction doesn't affect other transactions.

Durable If committed, the transaction results are written to persistent storage, such as the hard drive.

SQLite is an ACID-compliant database; it has even passed tests that simulate the computer losing power in the middle of a transaction, so you have high assurance that the database file won't be left in a corrupt, unusable state. A SQLite query will either completely insert data into the database or not insert it at all.

SQL Injection Attacks

A category of hacking techniques called *SQL injection attacks* can change your queries to do things you didn't intend. These techniques are beyond the scope of this book, and they mostly likely are not an issue for your code if your program isn't accepting data from strangers on the internet. But to prevent them, use the ? question mark syntax whenever you reference variables when inserting or updating data in your database.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

For example, if I want to insert a new cat record based on data stored in variables, I shouldn't insert these variables directly into the query string using Python, like this:

```
>>> cat_name = 'Zophie'
>>> cat_bday = '2021-01-24'
>>> fur_color = 'black'
>>> cat_weight = 5.6
>>> conn.execute(f'INSERT INTO cats VALUES ("{cat_name}", "{cat_bday}",
"{fur_color}", {cat_weight})')
<sqlite3.Cursor object at 0x0000022B91BB7C40>
```

If the values of these variables came from user input such as a web app form, a hacker could potentially specify strings that changed the meaning of the query. Instead, I should use a `?` in the query string, then pass the variables in a list argument following the query string:

```
>>> conn.execute(f'INSERT INTO cats VALUES (?, ?, ?, ?)', [cat_name, cat_bday,
fur_color, cat_weight])
<sqlite3.Cursor object at 0x0000022B91BB7C40>
```

The `execute()` method replaces the `?` placeholders in the query string with the variable values after making sure they won't cause a SQL injection attack. While such attacks are unlikely to apply to your code, it's a good habit to use the `?` placeholders instead of formatting the query string yourself.

Reading Data from the Database

Once there's data inside the database, you can read it with a `SELECT` query. Enter the following into the interactive shell to read data from the *example.db* database:

```
>>> import sqlite3
>>> conn = sqlite3.connect('example.db', isolation_level=None)
>>> conn.execute('SELECT * FROM cats').fetchall()
[('Zophie', '2021-01-24', 'black', 5.6)]
```

The `execute()` method call for the `SELECT` query returns a `Cursor` object. To obtain the actual data, we call the `fetchall()` method on this `Cursor` object. Each record is returned as a tuple in the list of tuples. The data in each tuple appears in the order of the table's columns.

Instead of writing Python code to sort through this list of tuples yourself, you can make SQLite extract the specific information you want. The example `SELECT` query has four parts:

- The `SELECT` keyword
- The columns you want to retrieve, where `*` means "all columns except `rowid`"
- The `FROM` keyword
- The table to retrieve data from; in this case, the `cats` table

If you wanted just the rowid and name columns of records in the cats table, your query would look like this:

```
>>> conn.execute('SELECT rowid, name FROM cats').fetchall()
[(1, 'Zophie')]
```

You can also use SQL to filter the query results, as you'll learn in the next section.

Looping over Query Results

The fetchall() method returns your SELECT query results as a list of tuples. A common coding pattern is to use this data in a for loop to perform some operation for each tuple. For example, download the *sweigartcats.db* file from *https://nostarch.com/automate-boring-stuff-python-3rd-edition*, then enter the following into the interactive shell to process its data:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> for row in conn.execute('SELECT * FROM cats'):
...     print('Row data:', row)
...     print(row[0], 'is one of my favorite cats.')
...
Row data: ('Zophie', '2021-01-24', 'gray tabby', 5.6)
Zophie is one of my favorite cats.
Row data: ('Miguel', '2016-12-24', 'siamese', 6.2)
Miguel is one of my favorite cats.
Row data: ('Jacob', '2022-02-20', 'orange and white', 5.5)
Jacob is one of my favorite cats.
--snip--
```

The for loop can iterate over the tuples of row data returned by conn.execute() without calling fetchall(), and the code in the body of the for loop can operate on each row individually, because the row variable populates with a tuple of row data from the query. The code can then access the columns using the tuple's integer index: index 0 for the name, index 1 for the birthdate, and so on.

Filtering Retrieved Data

Our SELECT queries have been retrieving every row in the table, but we might want just the rows that match some filter criteria. Using the *sweigartcats.db* file, add a WHERE clause to the SELECT statement to provide search parameters, such as having black fur:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('SELECT * FROM cats WHERE fur = "black"').fetchall()
[❶ ('Zophie', '2021-01-24', 'black', 5.6), ('Toby', '2021-05-17', 'black',
6.8), ('Thor', '2013-05-14', 'black', 5.2), ('Sassy', '2017-08-20', 'black',
7.5), ('Hope', '2016-05-22', 'black', 7.6)]
```

In this example, the `WHERE` clause `WHERE fur = "black"` will retrieve data only for records that have `"black"` in the `fur` column.

SQLite defines its own operators for use in the `WHERE` clause, but they're similar to Python's operators: `=`, `!=`, `<`, `>`, `<=`, `>=`, `AND`, `OR`, and `NOT`. Note that SQLite uses the `=` operator to mean "is equal to," while Python uses the `==` operator for that purpose. On either side of the operator, you can put a column name or a literal value.

The comparison will occur for each row in the table. For example, for `WHERE fur = "black"`, SQLite makes the following comparisons:

- Because `fur` is `'black'` and `'black' = 'black'` is true, SQLite includes the row at ❶ in the results.
- For the row `(2, 'Miguel', '2016-12-24', 'siamese', 6.2)`, `fur` is `'siamese'` and `'siamese' = 'black'` is false, so it doesn't include the row in the results.
- For the row `(3, 'Jacob', '2022-02-20', 'orange and white', 5.5)`, `fur` is `'orange and white'` and `'orange and white' = 'black'` is false, so it doesn't include the row in the results.

. . . and so on, for every row in the `cats` table.

Let's continue the previous example with a more complicated `WHERE` clause: `WHERE fur = "black" OR birthdate >= "2024-01-01"'`. Let's also use the `pprint.pprint()` function to "pretty print" the returned list:

```
>>> import pprint
>>> matching_cats = conn.execute('SELECT * FROM cats WHERE fur = "black"
OR birthdate >= "2024-01-01"').fetchall()
>>> pprint.pprint(matching_cats)
[('Zophie', '2021-01-24', 'black', 5.6),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Taffy', '2024-12-09', 'white', 7.0),
 ('Hollie', '2024-08-07', 'calico', 6.0),
 ('Lewis', '2024-03-19', 'orange tabby', 5.1),
 ('Thor', '2013-05-14', 'black', 5.2),
 ('Shell', '2024-06-16', 'tortoiseshell', 6.5),
 ('Jasmine', '2024-09-05', 'orange tabby', 6.3),
 ('Sassy', '2017-08-20', 'black', 7.5),
 ('Hope', '2016-05-22', 'black', 7.6)]
```

All of the cats in the resulting `matching_cats` list have either black fur or a birthdate after January 1, 2024. Note that the birthdate is just a string. While comparison operators like `>=` typically perform alphabetical comparisons on strings, they can also perform temporal comparisons, as long as the birthdate format is *YYYY-MM-DD*.

The `LIKE` operator lets you match just the start or end of a value, treating the percent sign (`%`)as a wildcard. For example, `name LIKE "%y"` matches all the names that end with 'y', while name LIKE "Ja%" matches all the names that start with 'Ja':

```
>>> conn.execute('SELECT rowid, name FROM cats WHERE name LIKE "%y"').fetchall()
[(5, 'Toby'), (11, 'Molly'), (12, 'Dusty'), (17, 'Mandy'), (18, 'Taffy'), (25, 'Rocky'), (27,
'Bobby'), (30, 'Misty'), (34, 'Mitsy'), (38, 'Colby'), (40, 'Riley'), (46, 'Ruby'), (65,
'Daisy'), (67, 'Crosby'), (72, 'Harry'), (77, 'Sassy'), (85, 'Lily'), (93, 'Spunky')]
>>> conn.execute('SELECT rowid, name FROM cats WHERE name LIKE "Ja%"').fetchall()
[(3, 'Jacob'), (49, 'Java'), (75, 'Jasmine'), (80, 'Jamison')]
```

You can also put percent signs at the start and end of a string to match text anywhere in the middle. For example, name LIKE %ob% matches all names that have 'ob' anywhere in them:

```
>>> conn.execute('SELECT rowid, name FROM cats WHERE name LIKE "%ob%"').fetchall()
[(3, 'Jacob'), (5, 'Toby'), (27, 'Bobby')]
```

The LIKE operator does a case-insensitive match, so name LIKE "%ob%" also matches '%OB%', '%Ob%', and '%oB%'. To do a case-sensitive match, use the GLOB operator and * as the wildcard characters:

```
>>> conn.execute('SELECT rowid, name FROM cats WHERE name GLOB "*m*"').fetchall()
[(4, 'Gumdrop'), (9, 'Thomas'), (44, 'Sam'), (63, 'Cinnamon'), (75, 'Jasmine'),
(79, 'Samantha'), (80, 'Jamison')]
```

While name LIKE "%m%" matches either a lowercase or uppercase *m*, name GLOB "*m*" matches only the lowercase *m*.

SQLite's wide set of operators and functionality rivals that of any full programming language. You can read more about it in the SQLite documentation at *https://www.sqlite.org/lang_expr.html.*

Ordering the Results

While you can always sort the list returned by fetchall() by calling Python's sort() method, it's easier to have SQLite sort the data for you by adding an ORDER BY clause to your SELECT query. For example, if I wanted to sort the cats by fur color, I could enter the following:

```
>>> import sqlite3, pprint
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> pprint.pprint(conn.execute('SELECT * FROM cats ORDER BY fur').fetchall())
[('Iris', '2017-07-13', 'bengal', 6.8),
 ('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Thor', '2013-05-14', 'black', 5.2),
--snip--
 ('Celine', '2015-04-18', 'white', 7.3),
 ('Daisy', '2019-03-19', 'white', 6.0)]
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If there is a WHERE clause in your query, the ORDER BY clause must come after it. You can also order the rows based on multiple columns. For example, if you want to first sort the rows by fur color and then sort the rows within each fur color by birthdate, run the following:

```
>>> cur = conn.execute('SELECT * FROM cats ORDER BY fur, birthdate')
>>> pprint.pprint(cur.fetchall())
[('Iris', '2017-07-13', 'bengal', 6.8),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Thor', '2013-05-14', 'black', 5.2),
 ('Hope', '2016-05-22', 'black', 7.6),
--snip--
 ('Ginger', '2020-09-22', 'white', 5.8),
 ('Taffy', '2024-12-09', 'white', 7.0)]
```

The ORDER BY clause lists the fur column first, followed by the birthdate column, separated by a comma. By default, these sorts are in ascending order: the smallest values come first, followed by larger values. To sort in descending order, add the DESC keyword after the column name. You can also use the ASC keyword to specify ascending order if you want your query to be explicit and readable. To practice using these keywords, enter the following into the interactive shell:

```
>>> cur = conn.execute('SELECT * FROM cats ORDER BY fur ASC, birthdate DESC')
>>> pprint.pprint(cur.fetchall())
[('Ruby', '2023-12-22', 'bengal', 5.0),
 ('Elton', '2020-05-28', 'bengal', 5.4),
 ('Iris', '2017-07-13', 'bengal', 6.8),
 ('Toby', '2021-05-17', 'black', 6.8),
 ('Sassy', '2017-08-20', 'black', 7.5),
--snip--
 ('Mitsy', '2015-05-29', 'white', 5.0),
 ('Celine', '2015-04-18', 'white', 7.3)]
```

The output lists the cats by fur color in ascending order (with 'bengal' coming before 'white'). Within each fur color, the cats are sorted by birthdate in descending order (with '2023-12-22' coming before '2020-05-28').

Limiting the Number of Results

If you're interested in viewing only the first few rows returned by your SELECT query, you might try to use Python list slices to limit the results. For example, use the [:3] slice to show only the first three rows in the cats table:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('SELECT * FROM cats').fetchall()[:3]  # This is inefficient.
[('Zophie', '2021-01-24', 'gray tabby', 5.6), ('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20', 'orange and white', 5.5)]
```

This code works, but it's inefficient; it first fetches all of the rows from the table, then discards everything except for the first three. It would be faster for your program to fetch just the first three rows from the database. You can do this with a LIMIT clause:

```
>>> conn.execute('SELECT * FROM cats LIMIT 3').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6), ('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20', 'orange and white', 5.5)]
```

This code runs faster than the code that fetches all the rows, especially for tables with a large number of rows. The LIMIT clause must come after the WHERE and ORDER BY clauses if your SELECT query includes them, as in the following example:

```
>>> conn.execute('SELECT * FROM cats WHERE fur="orange" ORDER BY birthdate LIMIT 4').fetchall()
[('Mittens', '2013-07-03', 'orange', 7.4), ('Piers', '2014-07-08', 'orange', 5.2),
('Misty', '2016-07-08', 'orange', 5.2), ('Blaze', '2023-01-16', 'orange', 7.4)]
```

There are a few other clauses you can add to your SELECT queries, but they are beyond the scope of this chapter. You can learn more about them in the SQLite documentation.

Creating Indexes for Faster Data Reading

In a previous section, we ran a SELECT query to find records based on matching names. You could speed up this search by creating an index on the name column. A *SQL index* is a data structure that organizes a column's data. As a result, queries with WHERE clauses that use these columns will perform better. The downside is that the index takes up a little bit more storage, so queries that insert or update data will be slightly slower, because SQLite must also update the data's index. If your database is large, and you read data from it more often than you insert or update its data, creating an index may be worthwhile. However, you should conduct testing to verify that the index actually improves performance.

To create indexes on, say, the names and birthdate columns in the cats table, run the following CREATE INDEX queries:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('CREATE INDEX idx_name ON cats (name)')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('CREATE INDEX idx_birthdate ON cats (birthdate)')
<sqlite3.Cursor object at 0x0000013EC121A040>
```

Indexes require names, and by convention, we name them after the column to which they apply, along with the idx_ prefix. Index names are global across the entire database, so if the database contains multiple tables with columns named birthdate, you may also want to include the table in the index name, like idx_cats_birthdate. To see all the indexes that exist for a table, check the built-in sqlite_schema table with a SELECT query:

```
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_name',), ('idx_birthdate',)]
```

If you change your mind or find that the indexes aren't improving performance, you can delete them with a DROP INDEX query:

```
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_birthdate',) ('idx_name',)]
>>> conn.execute('DROP INDEX idx_name')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type = "index" AND
tbl_name = "cats"').fetchall()
[('idx_birthdate',)]
```

For small databases with only a few thousand records, you can safely ignore indexes, as the benefits they provide are negligible. However, if you find that your database queries are taking a noticeable amount of time, creating indexes could improve their performance.

Updating Data in the Database

Once you've inserted rows into a table, you can change a row with an UPDATE statement. For example, let's update the record (1, 'Zophie', '2021-01-24', 'black', 5.6) to change the fur color from 'black' to 'gray tabby' in the *sweigartcats.db* file:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('SELECT * FROM cats WHERE rowid = 1').fetchall()
[('Zophie', '2021-01-24', 'black', 5.6)]
>>> conn.execute('UPDATE cats SET fur = "gray tabby" WHERE rowid = 1')
<sqlite3.Cursor object at 0x0000013EC121A040>
>>> conn.execute('SELECT * FROM cats WHERE rowid = 1').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6)]
```

The UPDATE statement has the following parts:

- The UPDATE keyword
- The name of the table containing the rows to update
- The SET clause, which specifies the column to update, as well as the value to update it to
- The WHERE clause, which specifies which rows to update

You can update multiple columns at a time by separating them with commas. For example, the query 'UPDATE cats SET fur = "black", weight = 6

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

WHERE rowid = 1' updates the value in the fur and weight columns to "black" and 6, respectively.

The UPDATE query updates every row in which the WHERE clause is true. If you ran the query 'UPDATE cats SET fur = "gray tabby" WHERE name = "Zophie"', the updates would apply for every cat named Zophie. That might be more cats than you intended! This is why, in most update queries, the WHERE clause uses the primary key from the rowid column to specify an individual record to update. The primary key uniquely identifies a row, so using it in the WHERE clause ensures that you update only the one row you intended.

It's a common bug to forget the WHERE clause when updating data. For example, if you wanted to do a find-and-replace to change every cat with 'white and orange' fur to 'orange and white' fur, you would run the following:

```
>>> conn.execute('UPDATE cats SET fur = "orange and white" WHERE fur = "white and orange"')
```

If you forgot to include the WHERE clause, the updates would apply to every row in the table. and suddenly all of your cats would have orange and white fur!

To avoid this bug, always include a WHERE clause in your UPDATE queries, even if you intend to apply a change to every row. In that case, you can use WHERE 1. Since 1 is the value that SQLite uses for a Boolean True, this tells SQLite to apply the change to every row. It may seem silly to have a superfluous WHERE 1 at the end of your query, but it lets you avoid dangerous bugs that could easily wipe out real data.

Deleting Data from the Database

You can delete rows from a table with a DELETE query. For example, to remove Zophie from the cats table, run the following on the *sweigartcats.db* file:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('SELECT rowid, * FROM cats WHERE rowid = 1').fetchall()
[(1, 'Zophie', '2021-01-24', 'gray tabby', 5.6)]
>>> conn.execute('DELETE FROM cats WHERE rowid = 1')
<sqlite3.Cursor object at 0x0000020322D183C0>
>>> conn.execute('SELECT * FROM cats WHERE rowid = 1').fetchall()
[]
```

The DELETE statement has the following parts:

- The DELETE FROM keywords
- The name of the table containing the rows to delete
- The WHERE clause, which specifies which rows to delete

As with the INSERT statement, it's vital to always include a WHERE clause in your DELETE statements; otherwise, you'll delete every row from the table. If you intend to delete every row, use WHERE 1 so that you can identify any DELETE statement without a WHERE clause as a bug.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Rolling Back Transactions

You may sometimes want to run several queries all together, or else not run those queries at all, but you won't know which you want to do until you've run at least some of the queries. One way to handle this situation is to begin a new transaction, execute the queries, and then either *commit* all of the queries to the database to complete the transaction or *roll them back* so that the database looks as if none of them were made.

Normally, a new transaction starts and finishes every time you call conn.execute() when connected to the SQLite database in WAL mode. However, you can also run a BEGIN query to start a new transaction; then, you can either complete the transaction by calling conn.commit() or undo all the queries by calling conn.rollback().

For example, let's add two new cats to the cats table, then roll back the transaction so that the table remains unchanged:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('BEGIN')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES ("Socks", "2022-04-04", "white", 4.2)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES ("Fluffy", "2022-10-30", "gray", 4.5)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.rollback()  # This undoes the INSERT statements.
>>> conn.execute('SELECT * FROM cats WHERE name = "Socks"').fetchall()
[]
>>> conn.execute('SELECT * FROM cats WHERE name = "Fluffy"').fetchall()
[]
```

The new cats, Socks and Fluffy, were not inserted into the database.

On the other hand, if you want to apply all of the queries you've run, call conn.commit() to commit the changes to the database:

```
>>> conn.execute('BEGIN')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES ("Socks", "2022-04-04", "white", 4.2)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.execute('INSERT INTO cats VALUES ("Fluffy", "2022-10-30", "gray", 4.5)')
<sqlite3.Cursor object at 0x00000219C8BF7C40>
>>> conn.commit()
>>> conn.execute('SELECT * FROM cats WHERE name = "Socks"').fetchall()
[('Socks', '2022-04-04', 'white', 4.2)]
>>> conn.execute('SELECT * FROM cats WHERE name = "Fluffy"').fetchall()
[('Fluffy', '2022-10-30', 'gray', 4.5)]
```

Now the cats Socks and Fluffy have records in the database.

Backing Up Databases

A friend of mine was once making changes to a database used by an e-commerce site specializing in collectible sports cards. She had to correct some naming mistakes in a few cards, and had just typed `UPDATE cards SET name = 'Chris Clemons'` when her cat walked on her keyboard, pressing ENTER. Without a `WHERE` clause, the query updated every one of the thousands of cards for sale on the website.

Fortunately, she had backups of the database, so she could restore it to its previous state. (This was especially useful because the same thing happened again in the exact same way, making her suspect the cat was doing it on purpose.)

If a program isn't currently accessing the SQLite database, you can back it up by simply copying the database file. A Python program might do this by calling `shutil.copy('sweigartcats.db', 'backup.db')`, as described in Chapter 11. If your software is constantly reading or updating the database's contents, however, you'll need to use the `Connection` object's `backup()` method instead. For example, enter the following into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> backup_conn = sqlite3.connect('backup.db', isolation_level=None)
>>> conn.backup(backup_conn)
```

The `backup()` method safely backs up the contents of the *sweigartcats.db* database to the *backup.db* file in between the other queries being run on it. Now that your data is safely backed up, your cat is free to step on your keyboard as much as it wants.

Altering and Dropping Tables

After creating a table in a database and inserting rows into it, you may want to rename the table or its columns. You may also wish to add or delete columns in the table, or even delete the entire table itself. You can use an `ALTER TABLE` query to perform these actions.

The following interactive shell examples start with a fresh copy of the *sweigartcats.db* database file. Run an `ALTER TABLE RENAME` query to rename the cats table to `felines`:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type="table"').fetchall()
[('cats',)]
>>> conn.execute('ALTER TABLE cats RENAME TO felines')  # Rename the table.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type="table"').fetchall()
[('felines',)]
```

To rename a column in a table, run an `ALTER TABLE RENAME COLUMN` query. For example, let's rename the `fur` column to `description`:

```
>>> conn.execute('PRAGMA TABLE_INFO(felines)').fetchall()[2]  # List the third column.
(2, 'fur', 'TEXT', 0, None, 0)
>>> conn.execute('ALTER TABLE felines RENAME COLUMN fur TO description')
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('PRAGMA TABLE_INFO(felines)').fetchall()[2]  # List the third column.
(2, 'description', 'TEXT', 0, None, 0)
```

To add a new column to the table, run an `ALTER TABLE ADD COLUMN` query. For example, let's add a new `is_loved` column to the `felines` table containing a Boolean value. SQLite uses `0` for false values and `1` for true values; we'll set the default value for `is_loved` to `1`:

```
>>> conn.execute('ALTER TABLE felines ADD COLUMN is_loved INTEGER DEFAULT 1')
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> import pprint
>>> pprint.pprint(conn.execute('SELECT * FROM felines LIMIT 3').fetchall())
[('Zophie', '2021-01-24', 'gray tabby', 5.6, 1),
 ('Miguel', '2016-12-24', 'siamese', 6.2, 1),
 ('Jacob', '2022-02-20', 'orange and white', 5.5, 1)]
```

It turns out the `is_loved` column isn't needed, since I store a `1` in it for all my cats, so I can remove the column with an `ALTER TABLE DROP COLUMN` query:

```
>>> conn.execute('PRAGMA TABLE_INFO(felines)').fetchall()  # List all columns.
[(0, 'name', 'TEXT', 1, None, 0), (1, 'birthdate', 'TEXT', 0, None, 0), (2, 'description',
'TEXT',
0, None, 0), (3, 'weight_kg', 'REAL', 0, None, 0), (4, 'is_loved', 'INTEGER', 0, '1', 0)]
>>> conn.execute('ALTER TABLE felines DROP COLUMN is_loved')  # Delete the column.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('PRAGMA TABLE_INFO(felines)').fetchall()  # List all columns.
[(0, 'name', 'TEXT', 1, None, 0), (1, 'birthdate', 'TEXT', 0, None, 0), (2, 'description',
'TEXT',
0, None, 0), (3, 'weight_kg', 'REAL', 0, None, 0)]
```

Any data stored in the deleted column will also be deleted.

If you want to delete the entire table, run a `DROP TABLE` query:

```
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type="table"').fetchall()
[('felines',)]
>>> conn.execute('DROP TABLE felines')  # Delete the entire table.
<sqlite3.Cursor object at 0x000001EDDB477C40>
>>> conn.execute('SELECT name FROM sqlite_schema WHERE type="table"').fetchall()
[]
```

Try to limit how often you change your tables and columns, as you'll also have to update the queries in your program to match.

Joining Multiple Tables with Foreign Keys

The structure of SQLite tables is rather strict; for example, each row has a set number of columns. But real-world data is often more complicated than a single table can capture. In relational databases, we can store complex data across multiple tables, and we can create links between them called *foreign keys*.

Say we want to store information about the vaccinations our cats receive. We can't just add columns to our cats table, as each cat could have one vaccination or many. Also, for each vaccination, we'd also want to list the vaccination date and the name of the doctor who administered it. SQL tables are not good at storing a list of columns. You would not want to have columns named vaccination1, vaccination2, vaccination3, and so on, for the same reason you wouldn't want variables named vaccination1 and vaccination2. If you create too many columns or variables, your code becomes a verbose, unreadable mess. If you create too few, you will have to constantly update your program to add more as needed.

Whenever you have a varying amount of data to add to a row, it makes more sense to list the added data as rows in a separate table, then have those rows refer back to the rows in the main table. In our *sweigartcats.db* database, add a second vaccinations table by entering the following into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> conn.execute('PRAGMA foreign_keys = ON')
<sqlite3.Cursor object at 0x000001E730AD03C0>
>>> conn.execute('CREATE TABLE IF NOT EXISTS vaccinations (vaccine TEXT,
date_administered TEXT, administered_by TEXT, cat_id INTEGER,
FOREIGN KEY(cat_id) REFERENCES cats(rowid)) STRICT')
<sqlite3.Cursor object at 0x000001CA42767D40>
```

The new vaccinations table has a column named cat_id with an INTEGER type. The integer values in this column matches the rowid values of the rows in the cats table. We call the cat_id column a *foreign key* because it refers to the primary key column of another table.

In the cats table, the cat Zophie has a rowid of 1. To record her vaccinations, we insert new rows into the vaccinations table with a cat_id value of 1:

```
>>> conn.execute('INSERT INTO vaccinations VALUES ("rabies", "2023-06-06", "Dr. Echo", 1)')
<sqlite3.Cursor object at 0x000001CA42767D40>
>>> conn.execute('INSERT INTO vaccinations VALUES ("FeLV", "2023-06-06", "Dr. Echo", 1)')
<sqlite3.Cursor object at 0x000001CA42767D40>
>>> conn.execute('SELECT * FROM vaccinations').fetchall()
[('rabies', '2023-06-06', 'Dr. Echo', 1), ('FeLV', '2023-06-06', 'Dr. Echo', 1)]
```

We could record vaccinations for other cats by using their rowid. If we wanted to add vaccination records for Mango, we could find Mango's rowid in the cats table and then add records to the vaccinations table using that value for the cat_id column:

```
>>> conn.execute('SELECT rowid, * FROM cats WHERE name = "Mango"').fetchall()
[(23, 'Mango', '2017-02-12', 'tuxedo', 6.8)]
>>> conn.execute('INSERT INTO vaccinations VALUES ("rabies", "2023-07-11", "Dr. Echo", 23)')
<sqlite3.Cursor object at 0x000001CA42767D40>
```

We can also perform a type of SELECT query called an *inner join*, which returns the linked rows from both tables. For example, enter the following into the interactive shell to retrieve the vaccinations rows joined with the data from the cats table:

```
>>> conn.execute('SELECT * FROM cats INNER JOIN vaccinations ON cats.rowid =
vaccinations.cat_id').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6, 'rabies', '2023-06-06', 'Dr. Echo', 1),
 ('Zophie', '2021-01-24', 'gray tabby', 5.6, 'FeLV', '2023-06-06', 'Dr. Echo', 1),
 ('Mango', '2017-02-12', 'tuxedo', 6.8, 'rabies', '2023-07-11', 'Dr. Echo', 23)]
```

Note that while you could make cat_id an INTEGER column and use it as a foreign key without actually setting up the FOREIGN KEY(cat_id) REFERENCES cats(rowid) syntax, foreign keys have several safety features to ensure that your data remains consistent. For example, you can't insert or update a vaccination record using a cat_id for a nonexistent cat. SQLite also forces you to delete all vaccination records for a cat before deleting the cat so as to not leave behind "orphaned" vaccination records.

These safety features are disabled by default. You can enable them by running the PRAGMA query after calling sqlite3.connect():

```
>>> conn.execute('PRAGMA foreign_keys = ON')
```

Foreign keys and joins have additional features, but they are outside the scope of this book.

In-Memory Databases and Backups

If your program is making a large number of queries, you can significantly improve the speed of your database by using an *in-memory database*. These databases are stored entirely in the computer's memory rather than in a file on the computer's hard drive. This makes changes incredibly fast. However, you'll need to remember to save the in-memory database to a file using the backup() method. If your program crashes in the middle of running, you'll lose the entire in-memory database, just as you would the values in the program's variables.

The following example creates an in-memory database and then saves it to a database in the file *test.db*:

```
>>> import sqlite3
>>> memory_db_conn = sqlite3.connect(':memory:',
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
isolation_level=None)  # Create an in-memory database.
>>> memory_db_conn.execute('CREATE TABLE test (name TEXT, number REAL)')
<sqlite3.Cursor object at 0x000001E730AD0340>
>>> memory_db_conn.execute('INSERT INTO test VALUES ("foo", 3.14)')
<sqlite3.Cursor object at 0x000001D9B0A07EC0>
>>> file_db_conn = sqlite3.connect('test.db', isolation_level=None)
>>> memory_db_conn.backup(file_db_conn)  # Save the database to test.db.
```

You can load a SQLite database file into memory with the backup() method as well:

```
>>> import sqlite3
>>> file_db_conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> memory_db_conn = sqlite3.connect(':memory:', isolation_level=None)
>>> file_db_conn.backup(memory_db_conn)
>>> memory_db_conn.execute('SELECT * FROM cats LIMIT 3').fetchall()
[('Zophie', '2021-01-24', 'gray tabby', 5.6), ('Miguel', '2016-12-24',
'siamese', 6.2), ('Jacob', '2022-02-20', 'orange and white', 5.5)]
```

There are some downsides to using in-memory databases. If your program crashes from an unhandled exception, you'll lose the database. You can mitigate this risk by wrapping your code in a try statement that catches any unhandled exceptions and then uses an except statement to save the file to a database. Chapter 4 covers exception handling with the try and except statements.

Copying Databases

You can obtain a copy of a database by calling the iterdump() method on the Connection object. This method returns an iterator that generates the text of the SQLite queries needed to re-create the database. You can use iterators in for loops or pass them to the list() function to convert them to a list of strings. For example, to get the SQLite queries needed to re-create the *sweigartcats.db* database, enter the following into the interactive shell:

```
>>> import sqlite3
>>> conn = sqlite3.connect('sweigartcats.db', isolation_level=None)
>>> with open('sweigartcats-queries.txt', 'w', encoding='utf-8') as fileObj:
...     for line in conn.iterdump():
...         fileObj.write(line + '\n')
```

This code creates a *sweigartcats-queries.txt* file with the following SQLite queries, which can re-create the database:

```
BEGIN TRANSACTION;
CREATE TABLE "cats" (name TEXT NOT NULL, birthdate TEXT, fur TEXT, weight_kg REAL) STRICT;
INSERT INTO "cats" VALUES('Zophie','2021-01-24','gray tabby',5.6);
INSERT INTO "cats" VALUES('Miguel','2016-12-24','siamese',6.2);
INSERT INTO "cats" VALUES('Jacob','2022-02-20','orange and white',5.5);
--snip--
INSERT INTO "cats" VALUES('Spunky','2015-09-04','gray',5.9);
INSERT INTO "cats" VALUES('Shadow','2021-01-18','calico',6.0);
COMMIT;
```

The text of these queries will almost certainly be larger than the original database, but the queries have the advantage of being human readable and easy to edit before copying and pasting into your Python code or into a SQLite app, as we'll cover next.

SQLite Apps

At times, you may want to investigate a SQLite database directly without having to write all of this extraneous Python code. You can do so by installing the `sqlite3` command, which runs from a terminal command line window and is documented at *https://sqlite.org/cli.html*.

On Windows, download the files labeled "A bundle of command line tools for managing SQLite database files" from *https://sqlite.org/download.html* and place the *sqlite3.exe* program in a folder on the system `PATH`. (See Chapter 12 for information about the `PATH` environment variable and terminal windows.) The `sqlite3` command is preinstalled on macOS. For UbuntuLinux, run `sudo apt install sqlite3` to install it.

Next, in a terminal window, run **`sqlite3 example.db`** to connect to the database in *example.db*. If this file doesn't exist, `sqlite3` creates this file with an empty database. You can enter SQL queries into this tool, though unlike the queries passed to `conn.execute()` in Python, they must end with a semicolon. Also, text values in SQLite must be enclosed in single quotes only; double quotes are not allowed.

For example, enter the following into the terminal window:

```
C:\Users\Al>sqlite3 example.db
SQLite version 3.xx.xx
Enter ".help" for usage hints.
sqlite> CREATE TABLE IF NOT EXISTS cats (name TEXT NOT NULL,
birthdate TEXT, fur TEXT, weight_kg REAL) STRICT;
sqlite> INSERT INTO cats VALUES ('Zophie', '2021-01-24', 'gray tabby', 4.7);
sqlite> SELECT * from cats;
Zophie|2021-01-24|gray tabby|4.7
```

As you can see in this example, the `sqlite3` command line tool provides a sort of SQLite interactive shell for you to enter queries at its `sqlite>` prompt. The `.help` command displays additional commands, such as `.tables` (which shows the tables in the database) and `.headers` (which lets you turn column headers on or off):

```
sqlite> .tables
cats
sqlite> .headers on
sqlite> SELECT * from cats;
name|birthdate|fur|weight_kg
Zophie|2021-01-24|gray tabby|4.7
```

If the command line tool is too sparse for you, there are also free, open source apps for displaying SQLite databases in a graphical user interface (GUI) on Windows, macOS, and Linux:

- DB Browser for SQLite (*https://sqlitebrowser.org*)
- SQLite Studio (*https://sqlitestudio.pl*)
- DBeaver Community (*https://dbeaver.io*)

While these GUI apps make it easier to work with SQLite databases, it's still worth learning the text-based syntax of SQLite queries.

Summary

Computers make it possible to deal with large amounts of data, but simply putting data into a text file, or even a spreadsheet, might not organize it well enough for you to make efficient use of it. SQL databases such as SQLite offer an advanced way to not only store large amounts of information but also retrieve the precise data you want through the SQL language.

SQLite is an impressive database, and Python comes with the `sqlite3` module in its standard library. SQLite's version of SQL is different from that used in other relational databases, but it's similar enough that learning SQLite provides a good introduction to databases in general.

SQLite databases live in a single file without a dedicated server. They can contain multiple tables (which you can think of as analogous to spreadsheets), and each table can contain multiple columns. To edit a table's values, you can perform the CRUD operations (for create, read, update, and delete) with the `INSERT`, `SELECT`, `UPDATE`, and `DELETE` queries. To change tables and columns themselves, you can use the `ALTER TABLE` and `DROP TABLE` queries. Lastly, foreign keys allow you to link records in multiple tables together using a technique called joins.

There's a lot more to SQLite and databases than can be covered in one chapter. If you'd like to learn more about SQL databases in general, I recommend *Practical SQL, 2nd Edition* (No Starch Press, 2022) by Anthony DeBarros.

Practice Questions

1. What Python instructions will obtain a `Connection` object for a SQLite database in a file named *example.db*?
2. What Python instruction will create a new table named `students` with `TEXT` columns named `first_name`, `last_name`, and `favorite_color`?
3. How do you connect to a SQLite database in WAL mode?
4. What's the difference between the `INTEGER` and `REAL` data types in SQLite?
5. What does strict mode add to a table?
6. What does the * in the query `'SELECT * FROM cats'` mean?

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

7. What does CRUD stand for?
8. What does ACID stand for?
9. What query adds new records to a table?
10. What query deletes records from a table?
11. What happens if you don't specify the `WHERE` clause in an `UPDATE` query?
12. What is an index? What code would create an index for a column named `birthdate` in a table named `cats`?
13. What is a foreign key?
14. How can you delete a table named `cats`?
15. What "filename" do you specify to create an in-memory database?
16. How can you copy a database to another database?

Practice Programs

For practice, write programs to do the following tasks.

Cat Vaccination Checker

Download the *sweigartcats.db* database of my cats from the book's resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*. Write a program that opens this database and lists all cats that don't have vaccines named `'rabies'`, `'FeLV'`, and `'FVRCP'`. Also, check the database for errors by finding all vaccines that were administered on a date before the cat's birthday.

Meal Ingredients Database

Write a program that creates two tables, one for meals and one for ingredients, using these SQL queries:

```
CREATE TABLE IF NOT EXISTS meals (name TEXT) STRICT
CREATE TABLE IF NOT EXISTS ingredients (name TEXT,
meal_id INTEGER, FOREIGN KEY(meal_id) REFERENCES meals
(rowid)) STRICT
```

Then, write a program that prompts the user for input. If the user enters `'quit'`, the program should exit. The user can also enter a new meal name, followed by a colon and a comma-delimited list of ingredients: `'meal:ingredient1,ingredient2'`. Save the meal and its ingredients in the `meals` and `ingredients` tables.

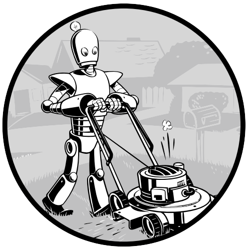
Finally, the user can enter the name of a meal or ingredient. If the name appears in the `meals` table, the program should list the meal's ingredients. If the name appears in the `ingredients` table, the program should list every meal that uses this ingredient. For example, the output of the program could look like this:

```
> onigiri:rice,nori,salt,sesame seeds
Meal added: onigiri
> chicken and rice:chicken,rice,cream of chicken soup
Meal added: chicken and rice
> onigiri
Ingredients of onigiri:
  rice
  nori
  salt
  sesame seeds
> chicken
Meals that use chicken:
  chicken and rice
> rice
Meals that use rice:
  onigiri
chicken and rice
> quit
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

17

PDF AND WORD DOCUMENTS



While you might think of PDF and Word as formats for storing text, these documents are binary files also containing font, color, and layout information, making them much more complex than simple plaintext files. If you want your programs to read or write PDFs or Word documents, you'll need to do more than simply pass their filenames to `open()`. Fortunately, several Python packages make these interactions easy. This chapter will cover two of them: PyPDF and Python-Docx.

PDF Documents

PDF stands for *Portable Document Format* and uses the *.pdf* file extension. Although PDFs support many features, this section will focus on three

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

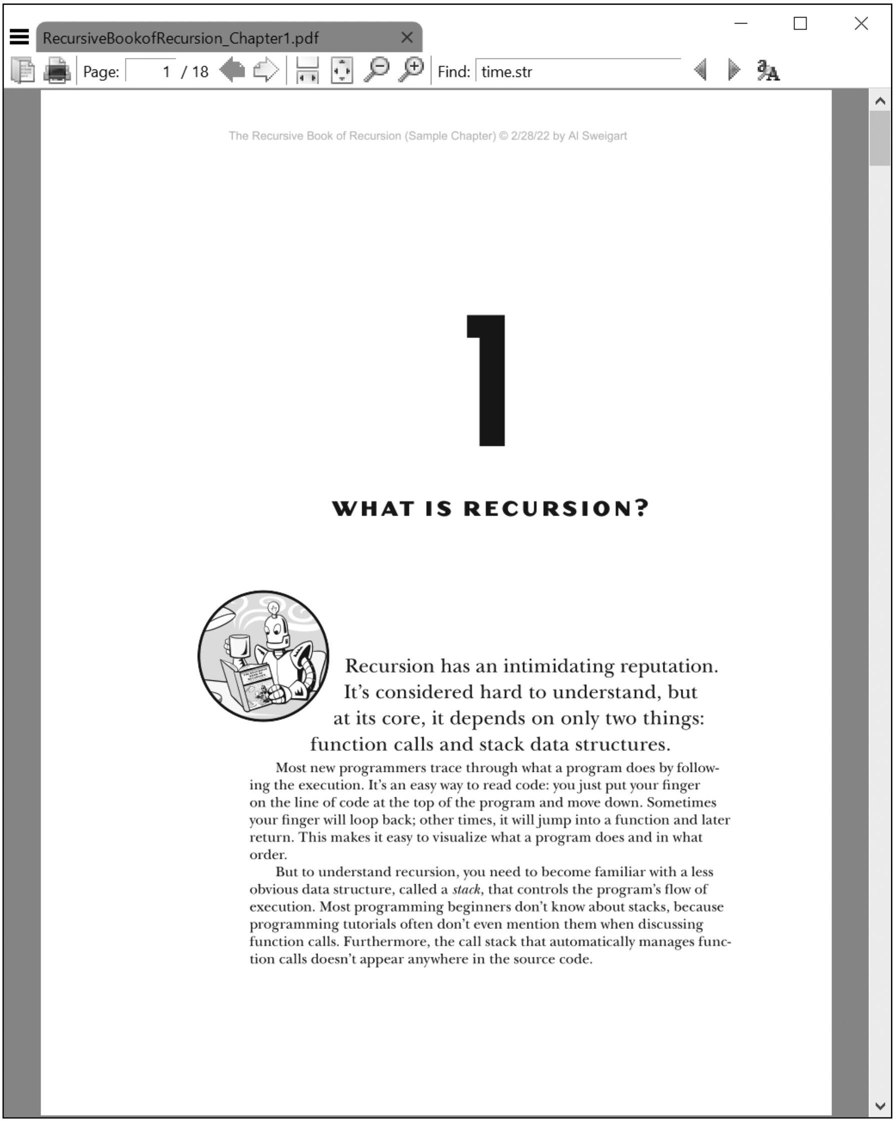
common tasks: extracting a document's text content, extracting its images, and crafting new PDFs from existing documents.

PyPDF is a Python package for creating and modifying PDF files. Install the package by following the instructions in Appendix A. If the package was installed correctly, running `import pypdf` in the interactive shell shouldn't display any errors.

While PDF files are great for laying out text in a way that is easy for people to print and read, they're not easy to parse into plaintext. As a result, PyPDF might make mistakes when extracting text from a PDF and may even fail to open some PDFs. There isn't much you can do about this, unfortunately. PyPDF may simply be unable to work with some of your particular files. That said, I haven't personally encountered a PDF file that PyPDF couldn't open.

Extracting Text

To begin working with PyPDF, let's use the PDF of a sample chapter from my book on recursive algorithms, *The Recursive Book of Recursion* (No Starch Press, 2022), shown in Figure 17-1.



The Recursive Book of Recursion (Sample Chapter) © 2/28/22 by Al Sweigart

1

WHAT IS RECURSION?

Recursion has an intimidating reputation. It's considered hard to understand, but at its core, it depends on only two things: function calls and stack data structures.

Most new programmers trace through what a program does by following the execution. It's an easy way to read code: you just put your finger on the line of code at the top of the program and move down. Sometimes your finger will loop back; other times, it will jump into a function and later return. This makes it easy to visualize what a program does and in what order.

But to understand recursion, you need to become familiar with a less obvious data structure, called a *stack*, that controls the program's flow of execution. Most programming beginners don't know about stacks, because programming tutorials often don't even mention them when discussing function calls. Furthermore, the call stack that automatically manages function calls doesn't appear anywhere in the source code.

Figure 17-1: The PDF file from which we will be extracting text

Download this *Recursion_Chapter1.pdf* file from the online resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*, then enter the following into the interactive shell:

```
  >>> import pypdf
❶ >>> reader = pypdf.PdfReader('Recursion_Chapter1.pdf')
❷ >>> len(reader.pages)
  18
```

Import the pypdf module, then call pypdf.PdfReader() with the filename of the PDF to get a PdfReader object that represents the PDF ❶. Store this object in a variable named reader.

The pages attribute of the PdfReader object is a list-like data structure of Page objects that represent individual pages in the PDF. Like actual Python lists, you can pass this data structure to the len() function ❷. This example PDF has 18 pages.

To extract the text from this PDF and output it to a text file, open a new file editor tab and save the following code to *extractpdftext.py*:

```
import pypdf
import pdfminer.high_level

PDF_FILENAME = 'Recursion_Chapter1.pdf'
TEXT_FILENAME = 'recursion.txt'

text = ''
try:
    reader = pypdf.PdfReader(PDF_FILENAME)
  ❶ for page in reader.pages:
      ❷ text += page.extract_text()
except Exception:
  ❸ text = pdfminer.high_level.extract_text(PDF_FILENAME)
with open(TEXT_FILENAME, 'w', encoding='utf-8') as file_obj:
  ❹ file_obj.write(text)
```

We use the pypdf module to extract the text, but if it fails for a particular PDF file and raises an exception, we fall back on the pdfminer module. Inside a try block, we use a for loop ❶ to iterate over each Page object in the PDF file's PdfReader object. Calling the Page object's extract_text() method ❷ returns a string that we can concatenate to the text variable. When the loop finishes, text will contain a single string of the entire text of the PDF.

If the PDF file has an unconventional format that PyPDF can't understand, we can try using pdfminer.high_level, an older module included in this book's third-party packages. The module's extract_text() function obtains the PDF's contents as a single string, rather than operating one page at a time ❸.

Finally, we can use the open() function and the write() method covered in Chapter 10 to write the string to a text file ❹.

Post-Processing with AI

The text extraction we just performed isn't perfect. The PDF file format is infamously convoluted and was originally designed for printing documents, not for making them machine readable. Even if there are no problems with the extraction, the text layout is fixed: the string will contain newline characters after each row of text and hyphenated words at the ends of rows. For instance, the extracted text from our example PDF looks like this:

```
1
WHAT IS RECURSION?
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
Recursion has an intimidating reputation.
It's considered hard to understand, but
at its core, it depends on only two things:
 function calls and stack data structures.
Most new programmers trace through what a program does by follow -
ing the execution. It's an easy way to read code: you just put your finger
--snip--
```

As you can see, there are many subjective decisions to make:

- Where should paragraphs in the PDF end and begin?
- Should page numbers, headers, and footers be included in the extracted text?
- How should tables of data in the PDF be converted to plaintext?
- How much whitespace should be included in the extract text?

Cleaning up this text is boring and cannot easily be automated with code. However, a large language model (LLM) AI such as ChatGPT can understand the context of the text well enough to produce a cleaned-up version automatically. Use a prompt such as the following before copying and pasting the extracted text:

> The following is text extracted from several pages of a PDF of a book on recursive algorithms. Clean up this text. By this, I mean put paragraphs on a single, separate line. Also remove the footer and header text from each page. Also get rid of the hyphens at the end of each line for words split up across the line. Do not make any spelling, grammar corrections, or rewording. Here is the text . . .

In a trial, this prompt produced the following text:

```
WHAT IS RECURSION?

Recursion has an intimidating reputation. It's considered hard
to understand, but at its core, it depends on only two things:
function calls and stack data structures. Most new programmers
trace through what a program does by following the execution.
It's an easy way to read code: you just put your finger...
```

A human must always review the output of any AI system. For example, the LLM removed the chapter number `1` from the start of the text, which wasn't my intention. You may have to refine the prompt to correct any misunderstandings.

If you don't have access to an LLM, the PyPDF documentation has a list of post-processing tips with code snippets at *https://pypdf.readthedocs.io/en/latest/user/post-processing-in-text-extraction.html.*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Extracting Images

PyPDF can also extract the images from a PDF document. Each `Page` object has an `images` attribute containing a list-like data structure of `Image` objects. We can write the bytes of these `Image` objects to an image file opened in `'wb'` (write-binary) mode. An `Image` object also has a `name` attribute that contains a string of the image's name. Here is code that extracts images from all pages of the sample chapter PDF. Open a new file editor tab and save the following code as *extractpdfimages.py*:

```
import pypdf
PDF_FILENAME = 'Recursion_Chapter1.pdf'

reader = pypdf.PdfReader(PDF_FILENAME)
❶ image_num = 0
❷ for i, page in enumerate(reader.pages):
    print(f'Reading page {i+1} - {len(page.images)} images found...')
    try:
      ❸ for image in page.images:
          ❹ with open(f'{image_num}_page{i+1}_{image.name}', 'wb') as file:
              ❺ file.write(image.data)
            print(f'Wrote {image_num}_page{i+1}_{image.name}...')
          ❻ image_num += 1
    except Exception as exc:
      ❼ print(f'Skipped page {i+1} due to error: {exc}')
```

The output of this program will look like this:

```
Reading page 1 - 7 images found...
Wrote 0_page1_Im0.jpg...
Wrote 1_page1_Im1.png...
--snip--
Reading page 7 - 1 images found...
Skipped page 7 due to error: not enough image data
--snip--
Reading page 17 - 0 images found...
Reading page 18 - 0 images found...
```

The images in a PDF document often have generic names, like *Im0.jpg* or *Im1.png*, so we use a variable counter named `image_num` ❶ along with the page number to assign them unique names. First, we loop over each `Page` object in the `pages` attribute of the `PdfReader` object. Recall that Python's `enumerate()` function ❷ returns integer indexes and the list item of the list-like object we pass it. Each `Page` object has an `images` attribute that we'll iterate over as well ❸.

Inside that second, nested `for` loop that iterates over the `Image` objects in the `images` attribute, we call `open()` and use an f-string to provide the filename ❹. This filename is made up of the integer in the `image_num` counter, the page number, and the string in the `name` attribute of the `Image` object. Because `i` starts at `0` while PDF page numbers start at 1, we use `i+1` to store

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

the page number. This name will include the file extension, such as *.png* or *.jpg*. We must also pass `'wb'` to the `open()` function call so that the file is opened in write-binary mode. The bytes of the image file are stored in the `Image` object's `data` attribute, which we pass to the `write()` method ❺. After writing an image, the code increments `image_num` by `1` ❻.

If some incompatibility between the PDF file and PyPDF causes a `Page` object's `images` attribute to raise an exception, our `try` and `except` statements can catch it and print a short error message ❼. This way, a problem on one page won't cause the entire program to crash.

Like text extraction, image extraction may be imperfect. For example, PyPDF failed to detect many of the images from the sample chapter PDF, and showed an error message instead. Meanwhile, you may be surprised that PyPDF extracts small, blank images used as background or spacers. When working with PDFs, you'll often require human review to ensure that the output is acceptable.

Creating PDFs from Other Pages

PyPDF's counterpart to `PdfReader` is `PdfWriter`, which can create new PDF files. But PyPDF cannot write arbitrary text to a PDF like Python can with plaintext files. Instead, PyPDF's PDF-writing capabilities are limited to copying, merging, cropping, and transforming pages from other PDFs into new ones. The code in this interactive shell example creates a copy of the sample chapter PDF with just the first five pages:

```
  >>> import pypdf
❶ >>> writer = pypdf.PdfWriter()
❷ >>> writer.append('Recursion_Chapter1.pdf', (0, 5))
  >>> with open('first_five_pages.pdf', 'wb') as file:
❸ ...     writer.write(file)
  ...
  (False, <_io.BufferedWriter name='first_five_pages.pdf'>)
```

First, we create a `PdfWriter` object by calling `pypdf.PdfWriter()` ❶. The `PdfWriter` object in the `writer` variable represents a blank PDF document with zero pages. Then, the `PdfWriter` object's `append()` method copies the first five pages from the sample chapter PDF, which we identify by the `'Recursion_Chapter1.pdf'` filename ❷. (Despite the identical name, the `PdfWriter` object's `append()` method differs from the `append()` list method.)

The second argument to this method is the tuple `(0, 5)`, which tells the `PdfWriter` object to copy pages starting at page index `0` (the first page in the `PdfWriter` object), up to but not including page index `5`. PyPDF considers index `0` to be the first page, even though PDF applications call it page 1.

Finally, to write the contents of the `PdfWriter` object to a PDF file, call `open()` with the filename and `'wb'` mode, and then pass the `File` object to the `write()` method of the `PdfWriter` object ❸. This should generate a new PDF file.

The tuple provided to `append()` can contain either two or three integers. If a third integer is provided, the method skips that number of pages.

Because this behavior matches the range() function, you could pass the two or three integers to list(range()) to see which pages the code would copy:

```
>>> list(range(0, 5))  # Passing (0, 5) makes append() copy these pages:
[0, 1, 2, 3, 4]
>>> list(range(0, 5, 2))  # Passing (0, 5, 2) makes append() copy these pages:
[0, 2, 4]
```

The append() method can also accept a list argument with page number integers for each page to append. For example, say we replace the code in the previous interactive shell example with this:

```
>>> writer.append('Recursion_Chapter1.pdf', [0, 1, 2, 3, 4])
```

This code would also copy the first five pages of the PDF document to the PdfWriter object. Note that append() interprets tuples and list arguments differently; the tuple (0, 5) tells append() to copy pages at index 0 up to but not including page index 5, but the list [0, 5] would tell append() to individually copy page index 0 and then copy page index 5. This difference in meaning between tuples and lists is unconventional, and you won't see it in other Python libraries, but it's part of PyPDF's design.

The append() method adds the copied pages to the end of the PdfWriter object. To insert copied pages before the end, call the merge() method instead. The merge() method has an additional integer argument that specifies where to insert the pages. For example, look at this code:

```
>>> writer.merge(2, 'Recursion_Chapter1.pdf', (0, 5))
```

This code copies the pages at index 0 up to but not including index 5 and inserts them where page index 2 (the third page) is in the PdfWriter object in writer. The original page at index 2, and all other pages, get shifted back after the inserted set of pages.

Rotating Pages

We can also rotate the pages of a PDF in 90-degree increments with the rotate() method of Page objects. Pass either 90, 180, or 270 as an argument to this method to rotate the page clockwise, and either -90, -180, or -270 to rotate the page counterclockwise. Rotating pages is useful if you have many PDFs that are, for whatever reason, already incorrectly rotated and you need to rotate them back, or else need to rotate only a few select pages in a PDF document. PDF apps often have rotation features that you can use to manually correct PDFs, but Python allows you to quickly apply rotations to many PDFs to automate this boring task.

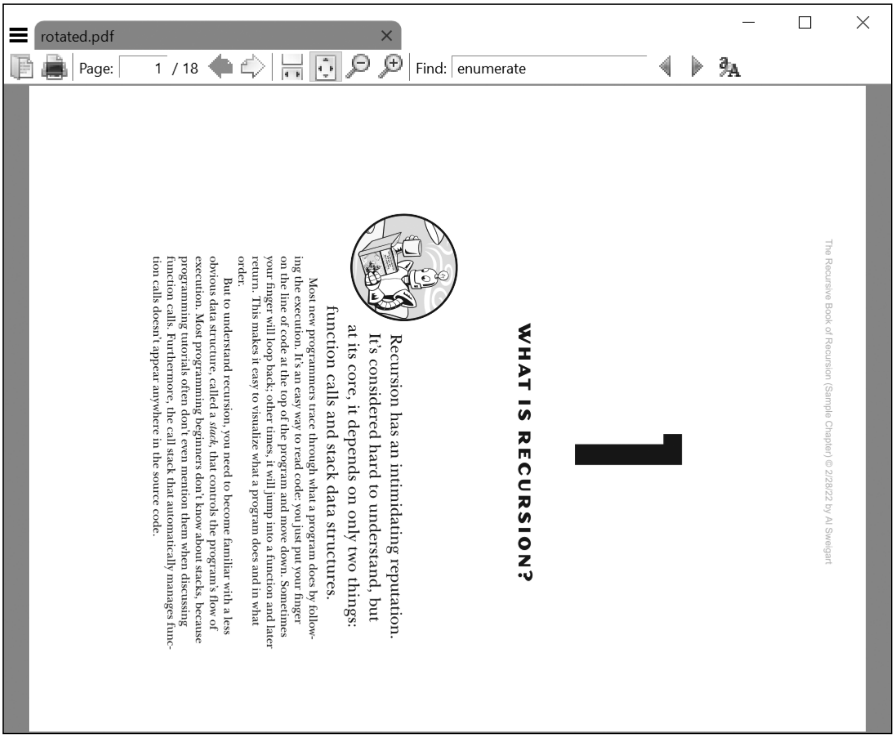
For example, enter the following into the interactive shell to rotate the pages of the sample chapter PDF:

```
>>> import pypdf
>>> writer = pypdf.PdfWriter()
```

```
  >>> writer.append('Recursion_Chapter1.pdf')
❶ >>> for i in range(len(writer.pages)):
  ...   ❷ writer.pages[i].rotate(90)
  ...
  {'/ArtBox': [21, 21, 525, 687], '/BleedBox': [12, 12, 534, 696],
  --snip--

  >>> with open('rotated.pdf', 'wb') as file:
  ...     writer.write(file)
  ...
  (False, <_io.BufferedWriter name='rotated.pdf'>)
```

We create a new `PdfWriter` object and copy the pages of the sample chapter PDF to it. Then, we use a `for` loop to loop over each page number. The call to `len(writer.pages)` returns the number of pages ❶ as an integer. The expression `writer.pages[i]` accesses each `Page` object on an iteration of the `for` loop, and the `rotate(90)` method call ❷ rotates this page in the `PdfWriter` object.

The resulting PDF should consist of all pages rotated 90 degrees clockwise, as shown in Figure 17-2.



The Recursive Book of Recursion (Sample Chapter) © 2/28/22 by Al Sweigart

1

WHAT IS RECURSION?

Recursion has an intimidating reputation. It's considered hard to understand, but at its core, it depends on only two things: function calls and stack data structures.

Most new programmers trace through what a program does by following the execution. It's an easy way to read code: you just put your finger on the line of code at the top of the program and move down. Sometimes your finger will loop back; other times, it will jump into a function and later return. This makes it easy to visualize what a program does and in what order.

But to understand recursion, you need to become familiar with a less obvious data structure, called a *stack*, that controls the program's flow of execution. Most programming beginners don't know about stacks, because programming tutorials often don't even mention them when discussing function calls. Furthermore, the call stack that automatically manages function calls doesn't appear anywhere in the source code.

Figure 17-2: The rotated.pdf *file with the page rotated 90 degrees clockwise*

PyPDF can't rotate documents in increments other than 90 degrees.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Inserting Blank Pages

You can insert or append a blank page to a `PdfWriter` object with the `insert_blank_page()` and `add_blank_page()` methods. The size of the new page will be the same as that of the preceding page. For example, let's create a copy of the sample chapter PDF with blank pages at the end and on page 3:

```
>>> import pypdf
>>> writer = pypdf.PdfWriter()
>>> writer.append('Recursion_Chapter1.pdf')
❶ >>> writer.add_blank_page()
{'/Type': '/Page', '/Resources': {}, '/MediaBox': [0.0, 0.0,
546, 708], '/Parent': IndirectObject(1, 0, 2629126028624)}
❷ >>> writer.insert_blank_page(index=2)
{'/Type': '/Page', '/Parent': NullObject, '/Resources': {},
'/MediaBox': RectangleObject([0.0, 0.0, 546, 708])}
>>> with open('with_blanks.pdf', 'wb') as file:
...     writer.write(file)  # Save the writer object to a PDFfile.
...
(False, <_io.BufferedWriter name='with_blanks.pdf'>)
```

After copying all the pages from the sample chapter PDF to the `PdfWriter` object, the `add_blank_page()` method adds a blank page to the end of the document. The `insert_blank_page()` method inserts a blank page at page index `2` (which is the third page, as page index `0` is the first page). This method requires that you specify the `index` parameter name.

You can either leave these pages blank or add content to them later, such as overlays and watermarks, as the next section explains.

Adding Watermarks and Overlays

PyPDF can also overlay the contents of one page on top of another, which is useful for adding a logo, timestamp, or watermark to a page. In PyPDF, a *stamp* or *overlay* is content placed on top of the page's existing content, while a *watermark* or *underlay* is content placed underneath the page's existing content.

Download *watermark.pdf* from the book's online resources and place the PDF in the current working directory along with the sample chapter PDF. Then, enter the following into the interactive shell:

```
>>> import pypdf
>>> writer = pypdf.PdfWriter()
>>> writer.append('Recursion_Chapter1.pdf')
❶ >>> watermark_page = pypdf.PdfReader('watermark.pdf').pages[0]
>>> for page in writer.pages:
❷ ...     page.merge_page(watermark_page, over=False)
...
>>> with open('with_watermark.pdf', 'wb') as file:
...     writer.write(file)
...
(False, <_io.BufferedWriter name='with_watermark.pdf'>)
```

This example creates a copy of the sample chapter PDF in a new PdfWriter object, saved in the writer variable. We also obtain the Page object for the first page of the watermark PDF and store it in the watermark_page variable. The for loop then loops over all the Page objects in the PdfWriter object and applies the watermark by passing it to merge_page(). (Don't confuse the merge_page() method of Page objects with the merge() method of PdfWriter objects discussed earlier in this chapter.)

The merge_page() method also has an over keyword argument. Pass True for this argument to create a stamp or overlay, or pass False to create a watermark or underlay.

After modifying the PdfWriter object's pages in the loop, the code then saves it as *with_watermark.pdf*. Figure 17-3 shows the original watermark PDF and two pages from the sample chapter PDF with the watermark applied.

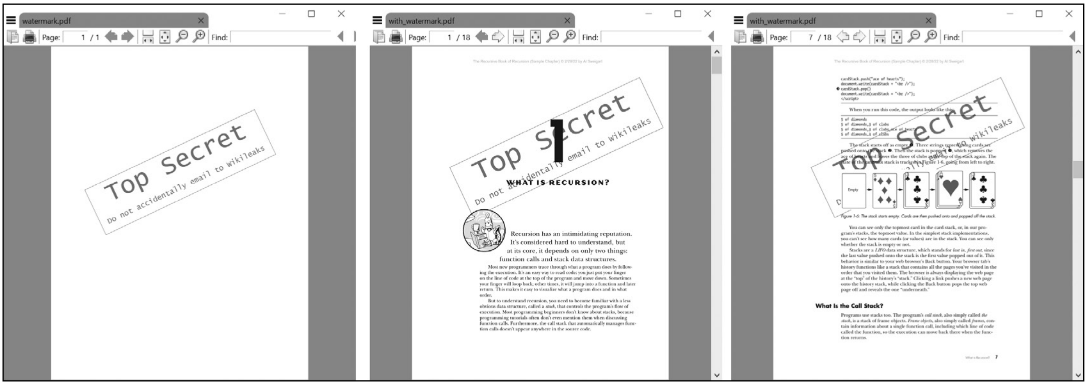


Figure 17-3: The watermark PDF (left) and pages with the added watermark (center, right)

The merge_page() method is useful for making broad changes to PDF documents, such as merging the contents of two pages.

Encrypting and Decrypting PDFs

PDFs allow you to encrypt their contents, making them unreadable. The encryption is only as strong as the password you choose, so create a password that uses different character types, isn't a word in the dictionary, and has around 14 to 16 characters. Keep in mind that PDFs have no password reset mechanism; if you forget the password, the PDF will be forever unreadable unless you can guess it.

The encrypt() method of PdfWriter objects accepts a password string and a string that selects the encryption algorithm. The 'AES-256' argument implements a recommended modern encryption algorithm, so we'll always use that. Enter the following into the interactive shell to create an encrypted copy of the sample chapter PDF:

```
>>> import pypdf
>>> writer = pypdf.PdfWriter()
>>> writer.append('Recursion_Chapter1.pdf')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
>>> writer.encrypt('swordfish', algorithm='AES-256')
>>> with open('encrypted.pdf', 'wb') as file:
...     writer.write(file)
...
(False, <_io.BufferedWriter name='encrypted.pdf'>)
```

The encrypt('swordfish', algorithm='AES-256') method call on the PdfWriter object encrypts the content of the PDF. After we write this encrypted PDF to the *encrypted.pdf* file, no PDF app, including PyPDF, should be able to open it without entering the password *swordfish*. (This is a poor password, as it's a word that occurs in the dictionary and is therefore easy to guess.) Encrypted data looks random unless you apply the correct decryption key or password, and decrypting the document with the wrong password results in garbage data. PDF apps will detect this, then prompt you to try the password again.

PyPDF can apply a password to an encrypted PDF to decrypt it. Enter the following into the interactive shell to detect encrypted PDFs with the is_encrypted attribute and decrypt them with decrypt():

```
  >>> import pypdf
❶ >>> reader = pypdf.PdfReader('encrypted.pdf')
  >>> writer = pypdf.PdfWriter()
❷ >>> reader.is_encrypted
  True
❸ >>> reader.pages[0]
  Traceback (most recent call last):
  --snip--
  pypdf.errors.FileNotDecryptedError: File has not been decrypted
❹ >>> reader.decrypt('an incorrect password').name
  'NOT_DECRYPTED'
❺ >>> reader.decrypt('swordfish').name
  'OWNER_PASSWORD'
❻ >>> writer.append(reader)
  >>> with open('decrypted.pdf', 'wb') as file:
  ...     writer.write(file)
  ...
  (False, <_io.BufferedWriter name='decrypted.pdf'>)
```

We load the encrypted PDF into a PdfReader object just like any other PDF ❶. The PdfReader object has an is_encrypted attribute ❷ that is set to either True or False. If you try to read the PDF content by, for example, accessing the pages attribute ❸, PyPDF raises a FileNotDecryptedError because it's unable to read it.

PDFs can have a *user password* that allows you to view the PDF and an *owner password* that allows you to set permissions for printing, commenting, extracting text, and other features. The user password and owner password are the first and second arguments to encrypt(), respectively. If only one string argument is passed to encrypt(), PyPDF will use it for both passwords.

To decrypt the PdfReader object, call the decrypt() method and pass it the string of the password. This method call returns a PasswordType object;

we're interested only in the `name` attribute of this object. If `name` is set to `'NOT_DECRYPTED'` ❹, we provided the wrong password. If `name` is set to `'OWNER_PASSWORD'` or `'USER_PASSWORD'` ❺, we've entered the correct owner or user password.

We can now append the pages from the `PdfReader` object to a `PdfWriter` object ❻ and save the decrypted PDF to a file.

Project 12: Combine Select Pages from Many PDFs

Say you have the boring job of merging several dozen PDF documents into a single PDF file. The first page of each document is a cover sheet, but you don't want the cover sheets repeated in the final result. Even though there are lots of free programs for combining PDFs, many of them simply merge entire files together. Let's write a Python program to customize the pages to include in the combined PDF.

At a high level, here is what the program will do:

- Find all PDF files in the current working directory and sort them alphabetically.
- For each PDF, copy all the pages after the first page to an output PDF.
- Save the output PDF to a file.

In terms of implementation, your code will need to do the following:

- Call `os.listdir()` to find all the files in the working directory and remove any non-PDF files. (We covered this function in Chapter 11.)
- Call Python's `sort()` list method to alphabetize the filenames.
- Create a `PdfWriter` object for the output PDF.
- Loop over each PDF file, creating a `PdfReader` object for it.
- From the `PdfReader` object, copy to the output PDF all the pages after the first page.
- Write the output PDF to a file.

Open a new file editor tab for this project and save it as *combine_pdfs.py*.

Step 1: Find All PDF Files

First, your program needs to get a list of all files with the *.pdf* extension in the current working directory and sort them. Make your code look like the following:

```
  # combine_pdfs.py - Combines all the PDFs in the current working directory
  # into a single PDF

❶ import pypdf, os

  # Get all the PDF filenames.
  pdf_filenames = []
  for filename in os.listdir('.'):
      if filename.endswith('.pdf'):
        ❷ pdf_files.append(filename)
  ❸ pdf_filenames.sort(key=str.lower)

  ❹ writer = pypdf.PdfWriter()

  # TODO: Loop through all the PDF files.

  # TODO: Copy all pages after the first page.

  # TODO: Save the resulting PDF to a file.
```

This code imports the `pypdf` and `os` modules ❶. The `os.listdir('.')` call will return a list of every file in the current working directory. The code then loops over this list, adding files with the *.pdf* extension to a list in the `pdf_filenames` variable ❷. Next, we sort this list in alphabetical order with the `key=str.lower` keyword argument to `sort()` ❸. For technical reasons, the `sort()` method puts uppercase characters like *Z* before lowercase characters like *a*; the keyword argument we provide prevents this by comparing the lowercase form of the strings. We create a `PdfWriter` object to hold the combined PDF pages ❹. Finally, a few comments outline the rest of the program.

Step 2: Open Each PDF

Now the program must read each PDF file in `pdf_filenames`. Add the following to your program:

```
# combine_pdfs.py - Combines all the PDFs in the current working directory
# into a single PDF

import pypdf, os

--snip--

# Loop through all the PDF files:
for pdf_filename in pdf_filenames:
    reader = pypdf.PdfReader(pdf_filename)
    # Copy all pages after the first page:
    writer.append(pdf_filename, (1, len(reader.pages)))

# TODO: Save the resulting PDF to a file.
```

For each PDF filename, the loop creates a `PdfFileReader` object and stores it in a variable named `reader`. Now the code inside the loop can call `len(reader.pages)` to find out how many pages the PDF has. It uses this information in the `append()` method call to copy pages starting at `1` (the second page, because PyPDF uses `0` as the first page index) up to the end of the PDF. Then, it appends the content to the same `PdfWriter` object in `writer`.

Step 3: Save the Results

Once these for loops have finished looping, the writer variable should contain a PdfFileWriter object with the pages of all the PDFs combined. The last step is to write this content to a file on the hard drive. Add this code to your program:

```
# combine_pdfs.py - Combines all the PDFs in the current working directory
# into a single PDF

import pypdf, os

--snip--

# Save the resulting PDF to a file:
with open('combined.pdf', 'wb') as file:
    writer.write(file)
```

Passing 'wb' to open() opens the output PDF file, *combined.pdf*, in write-binary mode. Then, passing the resulting File object to the write() method creates the actual PDF file. (Be aware of the identically named write() methods of File objects and PdfWriter objects.) At the end of the program, a single PDF contains all the pages (except the first) of every PDF in a folder, sorted alphabetically by filename.

Ideas for Similar Programs

Being able to create PDFs from the pages of other PDFs will let you make programs that can do the following:

- Cut out specific pages from PDFs.
- Reverse or reorder pages in a PDF.
- Create a PDF from only those pages of other PDFs that have some specific text, identified by the extract_text() method of Page objects.

Word Documents

Python can create and modify Microsoft Word documents, which have the *.docx* file extension, with the Python-Docx package, which you can install by following the instructions in Appendix A.

WARNING *Be sure to install Python-Docx, not Docx, which belongs to a different package that this book doesn't cover. When importing the module from the Python-Docx package, however, you'll need to run* import docx*, not* import python-docx*.*

If you don't have Word, you can use the free LibreOffice Writer application for Windows, macOS, and Linux to open *.docx* files. Download it from *https://www.libreoffice.org*. Although Word can run on macOS, this chapter will focus on Word for Windows. Also note that while the browser-based

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Office 365 and Google Docs web apps are popular word processors, they too import and export *.docx* files.

Compared to plaintext files, *.docx* files have many structural elements, which Python-Docx represents using three different data types. At the highest level, a `Document` object represents the entire document. The `Document` object contains a list of `Paragraph` objects for the paragraphs in the document. (A new paragraph begins whenever the user presses ENTER or RETURN while typing in a Word document.) Each of these `Paragraph` objects contains a list of one or more `Run` objects. The single-sentence paragraph in Figure 17-4 has four runs.

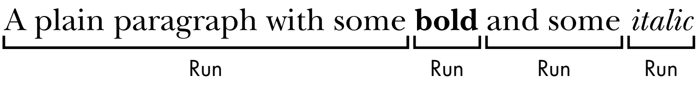


Figure 17-4: The `Run` objects identified in a `Paragraph` object

The text in a Word document is more than just a string. It has font, size, color, and other styling information associated with it. A *style* in Word is a collection of these attributes. A `Run` object is a contiguous run of text with the same style. You'll need a new `Run` object whenever the text style changes.

Reading Word Documents

Let's experiment with the `docx` module. Download *demo.docx* from the book's online resources and save the document to the working directory. Then, enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> len(doc.paragraphs)
7
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold text and some italic'
>>> len(doc.paragraphs[1].runs)
4
>>> doc.paragraphs[1].runs[0].text
'A plain paragraph with some '
>>> doc.paragraphs[1].runs[1].text
'bold'
>>> doc.paragraphs[1].runs[2].text
' and some '
>>> doc.paragraphs[1].runs[3].text
'italic'
```

We open a *.docx* file in Python, call `docx.Document()`, and pass it the filename *demo.docx*. This will return a `Document` object, which has a `paragraphs` attribute that is a list of `Paragraph` objects. When we call `len()` on this attribute, it returns 7, which tells us that there are seven Paragraph objects in this document. Each of these Paragraph objects has a text attribute that contains a string of the text in that paragraph (without the style information). Here, the first text attribute contains 'DocumentTitle', and the second contains 'A plain paragraph with some bold text and some italic'.

Each Paragraph object also has a runs attribute that is a list of Run objects. Run objects also have a text attribute, containing just the text in that particular run. Let's look at the text attributes in the second Paragraph object. Calling len() on this object tells us that there are four Run objects. The first Run object contains 'A plain paragraph with some '. Then, the text changes to a bold style, so 'bold' starts a new Run object. The text returns to an unbolded style after that, which results in a third Run object, ' text and some '. Finally, the fourth and last Run object contains 'italic' in an italic style.

Using Python-Docx, your Python programs can now read the text from a *.docx* file and use it just like any other string value.

Getting the Full Text from a .docx File

If you care only about a Word document's text and not about its styling information, you can use the getText() function. It accepts a filename of a *.docx* file and returns a single string value of its text. Open a new file editor tab and enter the following code, saving it as *readDocx.py*:

```
import docx

def get_text(filename):
    doc = docx.Document(filename)
    full_text = []
    for para in doc.paragraphs:
        full_text.append(para.text)
    return '\n'.join(full_text)
```

The get_text() function opens the Word document, loops over all the Paragraph objects in the paragraphs list, and then appends their text to the list in full_text. After the loop, the code joins the strings in full_text with newline characters.

You can import the *readDocx.py* program like any other module. Now, if you need just the text of a Word document, you can enter the following:

```
>>> import readDocx
>>> print(readDocx.get_text('demo.docx'))
Document Title
A plain paragraph with some bold text and some italic
Heading, level 1
Intense quote
first item in unordered list
first item in ordered list
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

You can also adjust `get_text()` to modify the string before returning it. For example, to indent each paragraph, replace the `append()` call in *readDocx.py* with this:

```
full_text.append('  ' + para.text)
```

To add a double space between paragraphs, change the `join()` call code to this:

```
return '\n\n'.join(full_text)
```

As you can see, it takes only a few lines of code to write functions that will read a *.docx* file and return a string of its content to your liking.

Styling Paragraph and Run Objects

Word and other word processors use styles to keep the visual presentation of text consistent and easy to change. For example, perhaps you want all body paragraphs to be 11-point, Times New Roman, left-justified, ragged-right text. You can create a style with these settings and assign it to all body paragraphs. If you later want to change the presentation of all body paragraphs in the document, you can change the style to automatically update those paragraphs.

To view styles in the browser-based Office 365 Word application, click the **Home** menu item, then the **Headings and Other Styles** drop-down menu, which will likely display "Normal" or another style name. Click **See More Styles** to bring up the More Styles window. In the Microsoft Word desktop application for Windows, you can see the styles by pressing CTRL-ALT-SHIFT-S to display the Styles pane, which looks like Figure 17-5. In LibreOffice Writer, you can view the Styles pane by clicking the **View ▸ Styles** menu item.

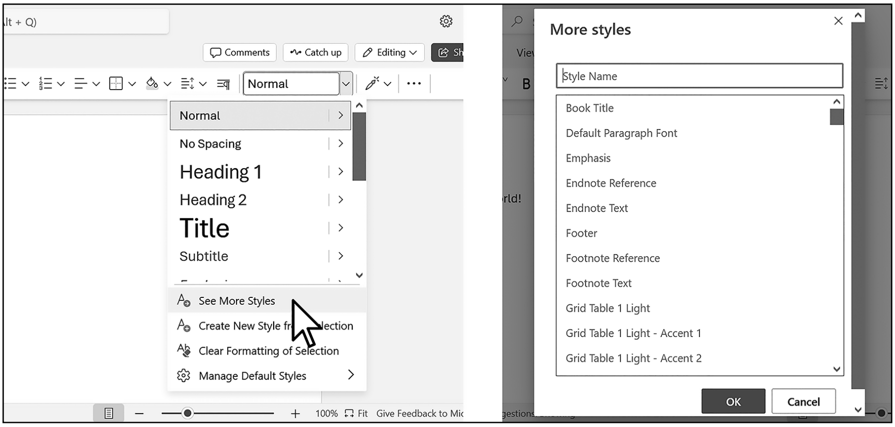


Figure 17-5: The Styles pane

Word documents contain three types of styles: paragraph styles apply to `Paragraph` objects, character styles apply to `Run` objects, and linked styles

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

apply to both kinds of objects. To style `Paragraph` and `Run` objects, set their `style` attribute to a string of the style's name. If `style` is set to `None`, no style will be associated with the `Paragraph` or `Run` object. The default Word styles have the following string values:

```
'Normal'            'Heading 5'         'List Bullet'       'List Paragraph'
'Body Text'         'Heading 6'         'List Bullet 2'     'MacroText'
'Body Text 2'       'Heading 7'         'List Bullet 3'     'No Spacing'
'Body Text 3'       'Heading 8'         'List Continue'     'Quote'
'Caption'           'Heading 9'         'List Continue 2'   'Subtitle'
'Heading 1'         'Intense Quote'     'List Continue 3'   'TOC Heading'
'Heading 2'         'List'              'List Number '      'Title'
'Heading 3'         'List 2'            'List Number 2'
'Heading 4'         'List 3'            'List Number 3'
```

When using a linked style for a `Run` object, you'll need to add `' Char'` to the end of its name. For example, to set the Quote linked style for a `Paragraph` object, you would use `paragraphObj.style = 'Quote'`, but for a `Run` object, you would use `runObj.style = 'Quote Char'`.

To create custom styles, use the Word application to define them, then read them from the `style` attribute of a `Paragraph` or `Run` object.

Applying Run Attributes

We can further style runs using `text` attributes. Each attribute can be set to one of three values: `True` (meaning the attribute is always enabled, no matter what other styles are applied to the run), `False` (meaning the attribute is always disabled), or `None` (which defaults to whatever the run's style is set to). Table 17-1 lists the `text` attributes that can be set on `Run` objects.

Table 17-1: Run Object text Attributes

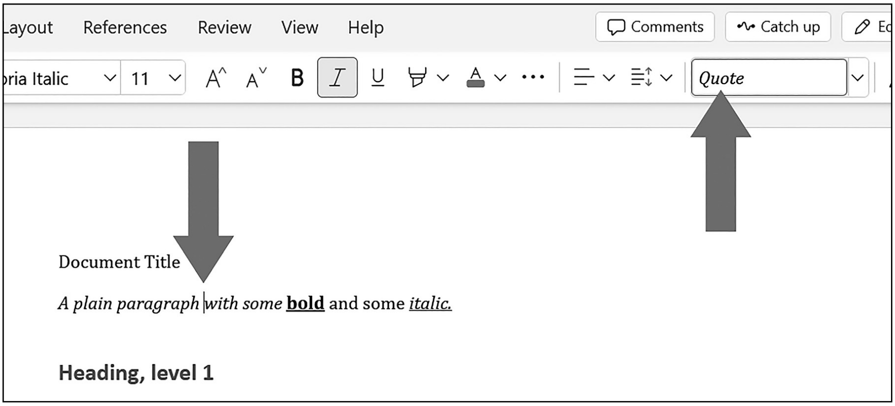
Attribute	Description
`bold`	The text appears in bold.
`italic`	The text appears in italic.
`underline`	The text is underlined.
`strike`	The text appears with a strikethrough.
`double_strike`	The text appears with a double strikethrough.
`all_caps`	The text appears in capital letters.
`small_caps`	The text appears in capital letters, with lowercase letters two points smaller.
`shadow`	The text appears with a shadow.
`outline`	The text appears outlined rather than solid.
`rtl`	The text is written right-to-left.
`imprint`	The text appears pressed into the page.
`emboss`	The text appears raised off the page in relief.

For example, to change the styles of *demo.docx*, enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document('demo.docx')
>>> doc.paragraphs[0].text
'Document Title'
>>> doc.paragraphs[0].style  # The exact id may be different:
_ParagraphStyle('Title') id: 3095631007984
>>> doc.paragraphs[0].style = 'Normal'
>>> doc.paragraphs[1].text
'A plain paragraph with some bold text and some italic'
>>> (doc.paragraphs[1].runs[0].text, doc.paragraphs[1].runs[1].text,
doc.paragraphs[1].runs[2].text, doc.paragraphs[1].runs[3].text)
('A plain paragraph with some ', 'bold', ' and some ', 'italic')
>>> doc.paragraphs[1].runs[0].style = 'Quote Char'
>>> doc.paragraphs[1].runs[1].underline = True
>>> doc.paragraphs[1].runs[3].underline = True
>>> doc.save('restyled.docx')
```

We use the text and style attributes to easily view the paragraphs in the document. As you can see, it's easy to divide a paragraph into runs and access each run individually. We get the first, second, and fourth runs in the second paragraph, style each run, and save the results to a new document.

Now the words *Document Title* at the top of *restyled.docx* should have the Normal style instead of the Title style, the Run object for the text *A plain paragraph with some* should have the Quote Char style, and the two Run objects for the words *bold* and *italic* should have their underline attributes set to True. Figure 17-6 shows how the styles of paragraphs and runs look in *restyled.docx*.



Document Title

A plain paragraph with some **bold** and some *italic.*

Heading, level 1

Figure 17-6: The restyled.docx *file*

You can find complete documentation on Python-Docx's use of styles at *https://python-docx.readthedocs.io/*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

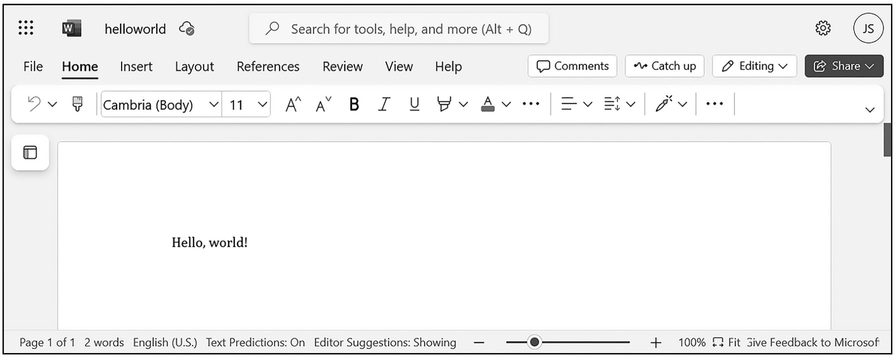
Writing Word Documents

To create your own *.docx* file, call `docx.Document()` to return a new, blank Word `Document` object. For example, enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello, world!')
<docx.text.paragraph.Paragraph object at 0x0000000003B56F60>
>>> doc.save('helloworld.docx')
```

The `add_paragraph()` document method adds a new paragraph of text to the document and returns a reference to the `Paragraph` object that was added. When you're done adding text, pass a filename string to the `save()` document method to save the `Document` object to a file.

This code will create a file named *helloworld.docx* in the current working directory. When opened, it should look like Figure 17-7. You can upload this *.docx* file into Office 365 or Google Docs or open it in Word or LibreOffice.



Hello, world!

Figure 17-7: The Word document created using `add_paragraph('Hello, world!')`

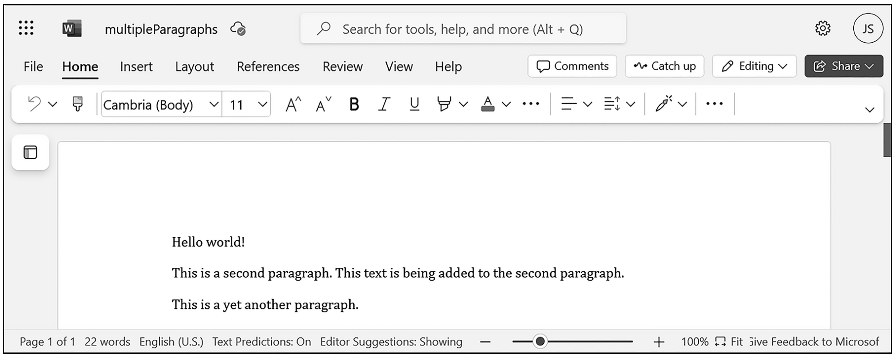
You can add paragraphs to the document by calling the `add_paragraph()` method again with the new paragraph's text. To add text to the end of an existing paragraph, call the paragraph's `add_run()` method and pass it a string. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_paragraph('Hello world!')
<docx.text.paragraph.Paragraph object at 0x000000000366AD30>
>>> para_obj_1 = doc.add_paragraph('This is a second paragraph.')
>>> para_obj_2 = doc.add_paragraph('This is a yet another paragraph.')
>>> para_obj_1.add_run(' This text is being added to the second paragraph.')
<docx.text.run.Run object at 0x0000000003A2C860>
>>> doc.save('multipleParagraphs.docx')
```

The resulting document should look like Figure 17-8. Note that the text *This text is being added to the second paragraph.* was added to the `Paragraph` object in `para_obj_1`, which was the second paragraph added to `doc`. The `add_paragraph()` and `add_run()` functions return `Paragraph` and `Run` objects, respectively, to save you the trouble of extracting them as a separate step.

Keep in mind that, as of this writing, new `Paragraph` objects can be added only to the end of the document, and new `Run` objects can be added only to the end of a `Paragraph` object.

Call the `save()` method again to save the additional changes you've made.


multipleParagraphs
Search for tools, help, and more (Alt + Q)
JS
File
Home
Insert
Layout
References
Review
View
Help
Comments
Catch up
Editing
Share
Cambria (Body)
11
Page 1 of 1 22 words English (U.S.) Text Predictions: On Editor Suggestions: Showing
100% Fit Give Feedback to Microsof

Hello world!

This is a second paragraph. This text is being added to the second paragraph.

This is a yet another paragraph.

Figure 17-8: The document with multiple `Paragraph` and `Run` objects added

Both `add_paragraph()` and `add_run()` accept an optional second argument that is a string of the `Paragraph` or `Run` object's style. Here is an example:

```
>>> doc.add_paragraph('Hello, world!', 'Title')
<docx.text.paragraph.Paragraph object at 0x00000213E6FA9190>
```

This line adds a paragraph with the text *Hello, world!* in the Title style.

Adding Headings

Calling `add_heading()` adds a paragraph with one of the heading styles. Enter the following into the interactive shell:

```
>>> import docx
>>> doc = docx.Document()
>>> doc.add_heading('Header 0', 0)
<docx.text.paragraph.Paragraph object at 0x00000000036CB3C8>
>>> doc.add_heading('Header 1', 1)
<docx.text.paragraph.Paragraph object at 0x00000000036CB630>
>>> doc.add_heading('Header 2', 2)
<docx.text.paragraph.Paragraph object at 0x00000000036CB828>
>>> doc.add_heading('Header 3', 3)
<docx.text.paragraph.Paragraph object at 0x00000000036CB2E8>
>>> doc.add_heading('Header 4', 4)
<docx.text.paragraph.Paragraph object at 0x00000000036CB3C8>
>>> doc.save('headings.docx')
```

The resulting *headings.docx* file should look like Figure 17-9.

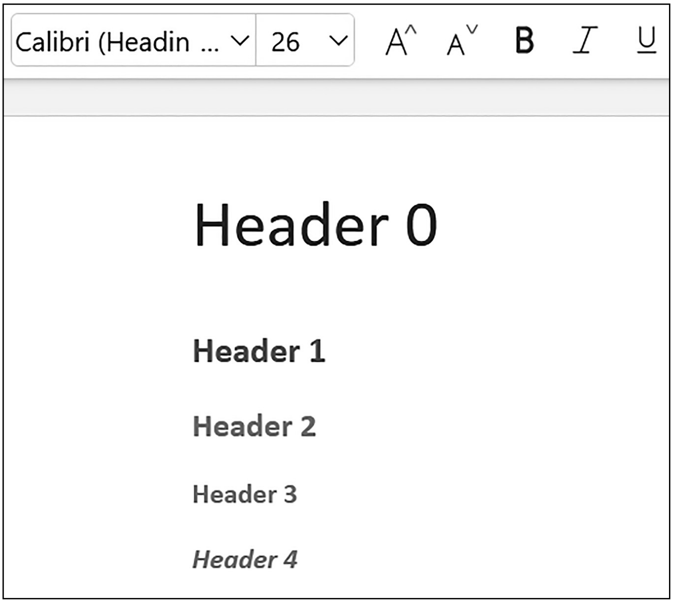


Figure 17-9: The headings.docx *document with headings 0 to 4*

The arguments to `add_heading()` are a string containing the heading text and an integer ranging from `0` to `4`. The integer `0` makes the heading the Title style, which we use for the top of the document. Integers `1` to `9` are for various heading levels, with `1` being the main heading and `9` being the lowest subheading. The `add_heading()` function returns a `Paragraph` object to save you the step of extracting it from the `Document` object as a separate step.

Adding Line and Page Breaks

To add a line break (rather than starting a whole new paragraph), you can call the `add_break()` method on the `Run` object you want to have the break appear after. If you want to add a page break instead, you need to pass the value `docx.enum.text.WD_BREAK.PAGE` as a lone argument to `add_break()`, as is done in the middle of the following example:

```
>>> doc = docx.Document()
>>> doc.add_paragraph('This is on the first page!')
<docx.text.paragraph.Paragraph object at 0x0000000003785518>
❶ >>> doc.paragraphs[0].runs[0].add_break(docx.enum.text.WD_BREAK.PAGE)
>>> doc.add_paragraph('This is on the second page!')
<docx.text.paragraph.Paragraph object at 0x00000000037855F8>
>>> doc.save('twoPage.docx')
```

This code creates a two-page Word document with *This is on the first page!* on the first page and *This is on the second page!* on the second. Even though there was still plenty of space on the first page after the text *This is on the first page!*, we forced the next paragraph to begin on a new page by inserting a page break after the first run of the first paragraph ❶.

Adding Pictures

You can use the `add_picture()` method of `Document` objects to add an image to the end of the document. Say you have a file *zophie.png* in the current working directory. You can add *zophie.png* to the end of your document with a width of 1 inch and height of 4 centimeters (Word can use both imperial and metric units) by entering the following:

```
>>> doc.add_picture('zophie.png', width=docx.shared.Inches(1), height=docx.shared.Cm(4))
<docx.shape.InlineShape object at 0x00000000036C7D30>
```

The first argument is a string of the image's filename. The optional `width` and `height` keyword arguments will set the width and height of the image in the document. If left out, the width and height will default to the normal size of the image.

You'll probably prefer to specify an image's height and width in familiar units such as inches and centimeters, so you can use the `docx.shared.Inches()` and `docx.shared.Cm()` functions when you're specifying the `width` and `height` keyword arguments.

Summary

Text information isn't just for plaintext files; in fact, it's pretty likely that you deal with PDFs and Word documents much more often. You can use the PyPDF package to read and write PDF documents, but many other Python libraries can read and write PDF files. If you want to go beyond those discussed in this chapter, I recommend searching for pdfplumber, ReportLab, pdfrw, PyMuPDF, pdfkit, and borb on the PyPI website.

Unfortunately, reading text from PDF documents might not always result in a perfect translation to a string, because the file format is complicated and some PDFs might not be readable at all. The pdfminer.six package is a fork of a no-longer-maintained pdfminer package that focuses on extracting text from PDFs. This chapter used pdfminer.six as a fallback mechanism if you're unable to extract text from a particular PDF file.

Word documents are more reliable, and you can read them with the python-docx package's `docx` module. You can manipulate text in Word documents via `Paragraph` and `Run` objects. These objects can also be given styles, though they must be from the default set of styles or from styles already in the document. You can add new paragraphs, headings, breaks, and pictures to the ends of documents.

Many of the limitations that come with working with PDFs and Word documents occur because these formats are meant to display nicely for human readers, rather than be easy to parse by software. The next chapter takes a look at some other common formats for storing information: CSV, JSON, XML, YAML, and TOML files. These formats were designed for use by computers, and you'll see that Python can work with them much more easily.

Practice Questions

1. What modes does the File object for PdfWriter objects need to be opened in to save the PDF file?
2. How do you acquire a Page object for page 5 from a PdfReader or PdfWriter object?
3. If a PdfReader object's PDF is encrypted with the password swordfish, what must you do before you can obtain Page objects from it?
4. If the rotate() method rotates pages clockwise, how do you rotate a page counterclockwise?
5. What method returns a Document object for a file named *demo.docx*?
6. What is the difference between a Paragraph object and a Run object?
7. How do you obtain a list of Paragraph objects for a Document object that's stored in a variable named doc?
8. What type of object has bold, underline, italic, strike, and outline variables?
9. What is the difference between setting the bold variable to True, False, or None?
10. How do you create a Document object for a new Word document?
11. How do you add a paragraph with the text 'Hello, there!' to a Document object stored in a variable named doc?
12. What integers represent the levels of headings available in Word documents?

Practice Programs

For practice, write programs to do the following tasks.

PDF Paranoia

Using the os.walk() function from Chapter 11, write a script that will go through every PDF in a folder (and its subfolders) and encrypt the PDFs using a password provided on the command line. Save each encrypted PDF with an *_encrypted.pdf* suffix added to the original filename. Before deleting the original file, have the program attempt to read and decrypt the new file to ensure that it was encrypted correctly.

Then, write a program that finds all encrypted PDFs in a folder (and its subfolders) and creates a decrypted copy of the PDF using a provided password. If the password is incorrect, the program should print a message to the user and continue to the next PDF.

Custom Invitations

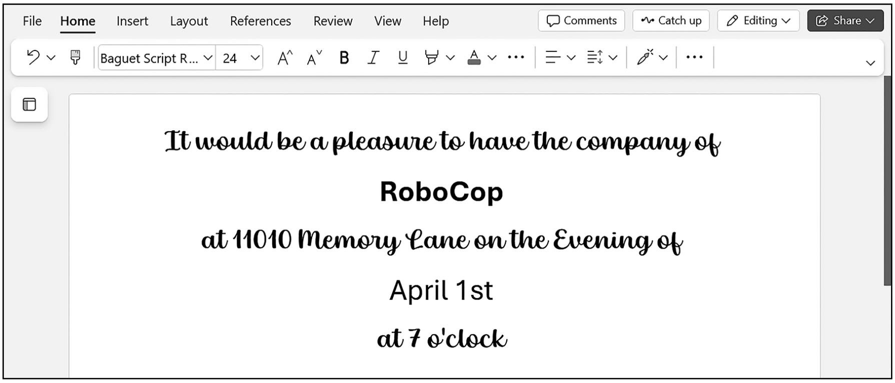
Say you have a text file of guest names. This *guests.txt* file has one name per line, as follows:

```
Prof. Plum
Miss Scarlet
```

```
Col. Mustard
Al Sweigart
RoboCop
```

Write a program that generates a Word document with custom invitations that look like Figure 17-10.

Because Python-Docx can only use styles that already exist in a Word document, you'll have to first add these styles to a blank Word file and then open that file with Python-Docx. There should be one invitation per page in the resulting Word document, so call `add_break()` to add a page break after the last paragraph of each invitation. This way, you will need to open only one Word document to print all of the invitations at once.



It would be a pleasure to have the company of

RoboCop

at 11010 Memory Lane on the Evening of

April 1st

at 7 o'clock

Figure 17-10: The Word document generated by your custom invite script

You can download a sample *guests.txt* file from the book's online resources.

PDF Password Breaker

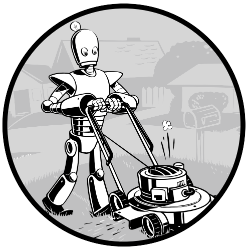
Say you have an encrypted PDF that you've forgotten the password to, but you remember it was a single English word. Trying to guess your forgotten password is quite a boring task. Instead, you can write a program that will decrypt the PDF by trying every possible English word until it finds one that works. This is called a *brute-force password attack*. Download the text file *dictionary.txt* from the book's online resources. This dictionary file contains over 44,000 English words, with one word per line.

Using the file-reading skills you learned in Chapter 10, create a list of word strings by reading this file. Then, loop over each word in this list, passing it to the `decrypt()` method. You should try both the uppercase and lowercase forms of each word. (On my laptop, going through all 88,000 uppercase and lowercase words from the dictionary file takes a couple of minutes. This is why you shouldn't use a simple English word for your passwords.)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

18

CSV, JSON, AND XML FILES



CSV, JSON, and XML are *data serialization* formats used to store data as plaintext files. Serialization converts data into a string to save your program's work to a text file, transfer it over an internet connection, or even just copy and paste it into an email. Python comes with the `csv`, `json`, and `xml` modules to help you work with these file formats.

While files in these formats are essentially text files that you could read and write with Python's `open()` function or the other file I/O functions from Chapter 10, it's easier to use Python's modules to handle them, just as we used the Beautiful Soup module in Chapter 13 to handle HTML-formatted text. Each format has its own use case:

Comma-separated values (CSV, pronounced "see-ess-vee") is a simplified spreadsheet format, and works best for storing a variable number of rows of data that share the same columns.

JavaScript Object Notation (JSON, pronounced "JAY-sawn" or "Jason") uses the same syntax as objects, arrays, and data types in the JavaScript programming language, though it doesn't require you to know how to program in JavaScript. It was created as a simpler alternative to XML.

Extensible Markup Language (XML, pronounced "ex-em-el") is an older, more established data serialization format widely used in enterprise software, but is overly complicated to work with if you don't need its advanced features.

This chapter covers the basics of these formats' syntax and the Python code to work with them.

The CSV Format

Each line in a CSV file (which uses the *.csv* file extension) represents a row in a spreadsheet, and commas separate the cells in the row. For example, the spreadsheet *example3.xlsx* included in the online resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* would look like this in a CSV file:

```
4/5/2035 13:34,Apples,73
4/5/2035 3:41,Cherries,85
4/6/2035 12:46,Pears,14
4/8/2035 8:59,Oranges,52
4/10/2035 2:07,Apples,152
4/10/2035 18:10,Bananas,23
4/10/2035 2:40,Strawberries,98
```

I'll use this file in this chapter's CSV interactive shell examples. Download it or enter the text into a text editor and save it as *example3.csv*.

You can think of CSV files as a list of lists of values. Python code could represent the *example3.csv* content as the value `[['4/5/2035 13:34', 'Apples', '73'], ['4/5/2035 3:41', 'Cherries', '85'], ... ['4/10/2035 2:40', 'Strawberries', '98']]`. CSV files are simple, lacking many of the features of an Excel spreadsheet. For example, they:

- Don't have multiple data types; every value is a string
- Don't have settings for font size or color
- Don't have multiple worksheets
- Can't specify cell widths or cell heights
- Can't merge cells
- Can't have embedded images or charts

The advantage of CSV files is simplicity. Many apps and programming languages support them, you can view them in text editors (including Mu), and they're a straightforward way to represent spreadsheet data.

Because CSV files are just text files, you might be tempted to read them as a string and then process that string using the techniques you learned in Chapter 8. For example, because each cell in a CSV file is separated by a comma, you might try to call split(',') on each line of text to get the comma-separated values as a list of strings. But not every comma in a CSV file represents the boundary between two cells. CSV files have a set of escape characters that allow you to include commas and other characters as part of the values. The split() method doesn't handle these escape characters. Because of these potential pitfalls, the csv module provides a more reliable way to read and write CSV files.

Reading CSV Files

To read a CSV file, you must create a csv.reader object, which lets you iterate over lines in the CSV file. The csv module comes with Python, so you can import it without having to first install it. Place *example3.csv* in the current working directory, then enter the following into the interactive shell:

```
>>> import csv
>>> example_file = open('example3.csv')
>>> example_reader = csv.reader(example_file)
>>> example_data = list(example_reader)
>>> example_data
[['4/5/2035 13:34', 'Apples', '73'], ['4/5/2035 3:41', 'Cherries', '85'],
['4/6/2035 12:46', 'Pears', '14'], ['4/8/2035 8:59', 'Oranges', '52'],
['4/10/2035 2:07', 'Apples', '152'], ['4/10/2035 18:10', 'Bananas', '23'],
['4/10/2035 2:40', 'Strawberries', '98']]
>>> example_file.close()
```

To read a CSV file with the csv module, open it using the open() function, just as you would any other text file, but instead of calling the read() or readlines() method on the File object that open() returns, pass it to the csv.reader() function. This function should return a reader object. Note that you can't pass a filename string directly to the csv.reader() function.

The easiest way to access the values in the reader object is to convert it to a plain Python list by passing it to list(). Using list() on this reader object returns a list of lists, which you can store in a variable, like example_data. Entering example_data in the shell displays the list of lists.

Now that you have the CSV file as a list of lists, you can access the value at a particular row and column with the expression example_data[row][col], where row is the index of one of the lists in example_data and col is the index of the item you want from that list. Enter the following into the interactive shell:

```
>>> example_data[0][0]  # First row, first column
'4/5/2035 13:34'
>>> example_data[0][1]  # First row, second column
'Apples'
>>> example_data[0][2]  # First row, third column
'73'
>>> example_data[1][1]  # Second row, second column
'Cherries'
>>> example_data[6][1]  # Seventh row, second column
'Strawberries'
```

As evident from the output, `example_data[0][0]` goes into the first list and gives us the first string, `example_data[0][2]` goes into the first list and gives us the third string, and so on.

Accessing Data in a for Loop

For large CSV files, you may want to use the `reader` object in a `for` loop. This approach saves you from having to load the entire file into memory at once. For example, enter the following into the interactive shell:

```
  >>> import csv
  >>> example_file = open('example3.csv')
  >>> example_reader = csv.reader(example_file)
❶ >>> for row in example_reader:
  ...   ❷ print('Row #' + str(example_reader.line_num) + ' ' + str(row))
  ...
  Row #1 ['4/5/2035 13:34', 'Apples', '73']
  Row #2 ['4/5/2035 3:41', 'Cherries', '85']
  Row #3 ['4/6/2035 12:46', 'Pears', '14']
  Row #4 ['4/8/2035 8:59', 'Oranges', '52']
  Row #5 ['4/10/2035 2:07', 'Apples', '152']
  Row #6 ['4/10/2035 18:10', 'Bananas', '23']
  Row #7 ['4/10/2035 2:40', 'Strawberries', '98']
```

After you import the `csv` module and make a `reader` object from the CSV file, you can loop through the rows in the `reader` object ❶. Each row is a list of values stored in the row variable, with each value in the list representing a cell.

The `print()` function call ❷ prints the number of the current row and the contents of the row. To get the row number, use the `reader` object's `line_num` attribute, which stores an integer. If your CSV file contains column headers in the first row, you could use `line_num` to check whether you're on row 1 and run a `continue` instruction to skip the headers. Unlike Python list indexes, line numbers in `line_num` begin at 1, not 0.

You can loop over the `reader` object only once. To reread the CSV file, you must call `open()` and `csv.reader()` again to create another `reader` object.

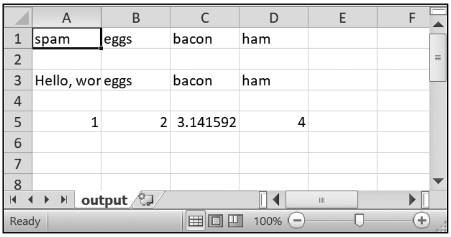
Writing CSV Files

A `csv.writer` object lets you write data to a CSV file. To create a `writer` object, use the `csv.writer()` function. Enter the following into the interactive shell:

```
  >>> import csv
❶ >>> output_file = open('output.csv', 'w', newline='')
❷ >>> output_writer = csv.writer(output_file)
  >>> output_writer.writerow(['spam', 'eggs', 'bacon', 'ham'])
  21
  >>> output_writer.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
  32
  >>> output_writer.writerow([1, 2, 3.141592, 4])
  16
  >>> output_file.close()
```

Call `open()` and pass it `'w'` to open a file in write mode ❶. This code should create an object you can then pass to `csv.writer()` ❷ to generate a `writer` object.

On Windows, you'll also need to pass a blank string for the `open()` function's `newline` keyword argument. For technical reasons that are beyond the scope of this book, if you forget to set the `newline` argument, the rows in *output.csv* will be double-spaced, as shown in Figure 18-1.



	A	B	C	D	E	F
1	spam	eggs	bacon	ham		
2						
3	Hello, wor	eggs	bacon	ham		
4						
5	1	2	3.141592	4		
6						
7						
8						

Figure 18-1: A double-spaced CSV file

The `writerow()` method of `writer` objects takes a list argument. Each value in the list will appear in its own cell in the output CSV file. The method's return value is the number of characters written to the file for that row (including newline characters). For example, this code in our example produces an *output.csv* file that looks like this:

```
spam,eggs,bacon,ham
"Hello, world!",eggs,bacon,ham
1,2,3.141592,4
```

Notice how the `writer` object automatically escapes the comma in the value `'Hello, world!'` with double quotes in the CSV file. The `csv` module saves you from having to handle these special cases yourself.

Using Tabs Instead of Commas

Tab-separated value (TSV) files are similar to CSV files but, unsurprisingly, use tabs instead of commas. Their files have the *.tsv* file extension. Say you want to separate cells with a tab character instead of a comma and want the rows to be double-spaced. You could enter something like the following into the interactive shell:

```
>>> import csv
>>> output_file = open('output.tsv', 'w', newline='')
>>> output_writer = csv.writer(output_file, delimiter='\t', lineterminator='\n\n') ❶
>>> output_writer.writerow(['spam', 'eggs', 'bacon', 'ham'])
21
>>> output_writer.writerow(['Hello, world!', 'eggs', 'bacon', 'ham'])
30
>>> output_writer.writerow([1, 2, 3.141592, 4])
16
>>> output_file.close()
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

This code changes the delimiter and line terminator characters in your file. The *delimiter* is the character that appears between cells on a row. By default, the delimiter for a CSV file is a comma. The *line terminator* is the character that comes at the end of a row. By default, the line terminator is a newline. You can change characters to different values by using the `delimiter` and `lineterminator` keyword arguments with `csv.writer()`.

Passing `delimiter='\t'` and `lineterminator='\n\n'` ❶ changes the delimiter to a tab and the line terminator to two newlines. The code then calls `writerow()` three times to create three row, producing a file named *output.tsv* with the following contents:

```
spam    eggs    bacon   ham

Hello, world!   eggs    bacon   ham

1   2   3.141592    4
```

Tabs now separate the cells in the spreadsheet.

Handling Header Rows

For CSV files that contain header rows, it's often more convenient to work with the `DictReader` and `DictWriter` objects rather than the `reader` and `writer` objects. While `reader` and `writer` read and write to CSV file rows by using lists, `DictReader` and `DictWriter` perform the same functions using dictionaries, treating the values in the first row as the keys.

Download *exampleWithHeader3.csv* from the book's online resources for the next example. This file is the same as *example3.csv* except it includes *Timestamp*, *Fruit*, and *Quantity* as column headers in the first row. To read the file, enter the following into the interactive shell:

```
>>> import csv
>>> example_file = open('exampleWithHeader3.csv')
>>> example_dict_reader = csv.DictReader(example_file)
❶ >>> example_dict_data = list(example_dict_reader)
>>> example_dict_data
[{'Timestamp': '4/5/2035 3:41', 'Fruit': 'Cherries', 'Quantity': '85'},
{'Timestamp': '4/6/2035 12:46', 'Fruit': 'Pears', 'Quantity': '14'},
{'Timestamp': '4/8/2035 8:59', 'Fruit': 'Oranges', 'Quantity': '52'},
{'Timestamp': '4/10/2035 2:07', 'Fruit': 'Apples', 'Quantity': '152'},
{'Timestamp': '4/10/2035 18:10', 'Fruit': 'Bananas', 'Quantity': '23'},
{'Timestamp': '4/10/2035 2:40', 'Fruit': 'Strawberries', 'Quantity': '98'}]
>>> example_file = open('exampleWithHeader3.csv')
>>> example_dict_reader = csv.DictReader(example_file)
❷ >>> for row in example_dict_reader:
...     print(row['Timestamp'], row['Fruit'], row['Quantity'])
...
4/5/2035 13:34 Apples 73
4/5/2035 3:41 Cherries 85
4/6/2035 12:46 Pears 14
4/8/2035 8:59 Oranges 52
```

```
4/10/2035 2:07 Apples 152
4/10/2035 18:10 Bananas 23
4/10/2035 2:40 Strawberries 98
```

By passing the DictReader object to list() ❶, you can get the CSV data as a list of dictionaries. Each row corresponds to one dictionary in the list. Alternatively, you can use the DictReader object inside a for loop ❷. The DictReader object sets row to a dictionary object with keys derived from the headers in the first row. Using a DictReader object means you don't need additional code to skip the first row's header information, as the DictReader object does this for you.

If you tried to use a DictReader object with *example3.csv*, which doesn't have column headers in the first row, the DictReader object would use '4/5/2035 13:34', 'Apples', and '73' as the dictionary keys. To avoid this, you can supply the DictReader() function with a second argument containing made-up header names:

```
>>> import csv
>>> example_file = open('example3.csv')
>>> example_dict_reader = csv.DictReader(example_file, ['time', 'name', 'amount'])
>>> for row in example_dict_reader:
...     print(row['time'], row['name'], row['amount'])
...
4/5/2035 13:34 Apples 73
4/5/2035 3:41 Cherries 85
4/6/2035 12:46 Pears 14
4/8/2035 8:59 Oranges 52
4/10/2035 2:07 Apples 152
4/10/2035 18:10 Bananas 23
4/10/2035 2:40 Strawberries 98
```

Because *example3.csv*'s first row doesn't contain column headings, we created our own: 'time', 'name', and 'amount'. The DictWriter objects use dictionaries to create CSV files:

```
>>> import csv
>>> output_file = open('output.csv', 'w', newline='')
>>> output_dict_writer = csv.DictWriter(output_file, ['Name', 'Pet', 'Phone'])
>>> output_dict_writer.writeheader()
16
>>> output_dict_writer.writerow({'Name': 'Alice', 'Pet': 'cat', 'Phone': '555-1234'})
20
>>> output_dict_writer.writerow({'Name': 'Bob', 'Phone': '555-9999'})
15
>>> output_dict_writer.writerow({'Phone': '555-5555', 'Name': 'Carol', 'Pet': 'dog'})
20
>>> output_file.close()
```

If you want your file to contain a header row, write that row by calling writeheader(). Otherwise, skip calling writeheader() to omit a header row from the file. You can then write each row of the CSV file with a writerow()

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

method call, passing a dictionary that uses the headers as keys and contains the data to write to the file.

The *output.csv* file that this code creates looks like this:

```
Name,Pet,Phone
Alice,cat,555-1234
Bob,,555-9999
Carol,dog,555-5555
```

The double commas indicate that Bob has a blank value for a pet. Notice that the order of the key-value pairs in the dictionaries you passed to `writerow()` doesn't matter; they're written in the order of the keys given to `DictWriter()`. For example, even though you passed the `Phone` key and value before the `Name` and `Pet` keys and values in the fourth row, the phone number still appears last in the output.

Notice also that any missing keys, such as `'Pet'` in `{'Name': 'Bob', 'Phone': '555-9999'}`, will become empty cells in the CSV file.

Project 13: Remove the Header from CSV Files

Say you have the boring job of removing the first line from several hundred CSV files. Maybe you'll be feeding them into an automated process that requires just the data, without the headers at the top of the columns. You *could* open each file in Excel, delete the first row, and resave the file—but that would take hours. Let's write a program to do it instead.

The program will need to open every file with the *.csv* extension in the current working directory, read the contents of the CSV file, and rewrite the contents without the first row to a file of the same name. This will replace the old contents of the CSV file with the new, headless contents.

WARNING *As always, whenever you write a program that modifies files, be sure to back up the files first, in case your program doesn't work the way you expect it to. You don't want to accidentally erase your original files.*

At a high level, the program must do the following:

- Find all the CSV files in the current working directory.
- Read the full contents of each file.
- Write the contents, skipping the first line, to a new CSV file.

At the code level, this means the program will need to do the following:

- Loop over a list of files from `os.listdir()`, skipping the non-CSV files.
- Create a CSV `reader` object and read the contents of the file, using the `line_num` attribute to figure out which line to skip.
- Create a CSV `writer` object and write the read-in data to the new file.

For this project, open a new file editor window and save it as *removeCsvHeader.py*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Step 1: Loop Through Each File

The first thing your program needs to do is loop over a list of all CSV filenames for the current working directory. Make *removeCsvHeader.py* look like this:

```
# Removes the header line from csv files
import csv, os

os.makedirs('headerRemoved', exist_ok=True)

# Loop through every file in the current working directory.
for csv_filename in os.listdir('.'):
    if not csv_filename.endswith('.csv'):
      ❶ continue  # Skip non-CSV files.

    print('Removing header from ' + csv_filename + '...')

    # TODO: Read the CSV file (skipping the first row).

    # TODO: Write the CSV file.
```

The `os.makedirs()` call create a *headerRemoved* folder in which to save the headless CSV files. A `for` loop on `os.listdir('.')` gets you partway there, but it will loop over *all* files in the working directory, so you'll need to add some code at the start of the loop that skips filenames that don't end with *.csv*. The `continue` statement ❶ makes the `for` loop move on to the next filename when it comes across a non-CSV file.

To see output as the program runs, print a message indicating which CSV file the program is working on. Then, add some `TODO` comments indicating what the rest of the program should do.

Step 2: Read the File

The program doesn't remove the first line from the CSV file. Rather, it creates a new copy of the CSV file without the first line. That way, we can use the original file in case a bug incorrectly modifies the new file.

The program will need a way to track whether it's currently looping on the first row. Add the following to *removeCsvHeader.py*.

```
# Removes the header line from csv files
import csv, os

--snip--

    # Read the CSV file (skipping the first row).
    csv_rows = []
    csv_file_obj = open(csv_filename)
    reader_obj = csv.reader(csv_file_obj)
    for row in reader_obj:
        if reader_obj.line_num == 1:
            continue  # Skip the first row.
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
        csv_rows.append(row)
    csv_file_obj.close()

    # TODO: Write the CSV file.
```

The reader object's line_num attribute can be used to determine which line in the CSV file it's currently reading. Another for loop will loop over the rows returned from the CSV reader object, and all rows but the first will be appended to csv_rows.

As the for loop iterates over each row, the code checks whether reader_obj.line_num is set to 1. If so, it executes a continue to move on to the next row without appending it to csv_rows. For every subsequent row, the condition will be always be False, and the code will append the row to csv_rows.

Step 3: Write the New CSV File

Now that csv_rows contains all rows but the first row, we need to write the list to a CSV file in the *headerRemoved* folder. Add the following to *removeCsvHeader.py*:

```
# Removes the header line from csv files
import csv, os

--snip--

# Loop through every file in the current working directory.
❶ for csv_filename in os.listdir('.'):
    if not csv_filename.endswith('.csv'):
        continue    # Skip non-CSV files.

    --snip--

    # Write the CSV file.
    csv_file_obj = open(os.path.join('headerRemoved', csv_filename), 'w',
                 newline='')
    csv_writer = csv.writer(csv_file_obj)
    for row in csv_rows:
        csv_writer.writerow(row)
    csv_file_obj.close()
```

The CSV writer object will write the list to a CSV file in headerRemoved using csv_filename (which we also used in the CSV reader). After creating the writer object, we loop over the sublists stored in csv_rows and write each sublist to the file.

The outer for loop ❶ will then loop to the next filename returned by os.listdir('.'). When that loop is finished, the program will be complete.

To test your program, download *removeCsvHeader.zip* from the book's online resources and unzip it to a folder. Then, run the *removeCsvHeader.py* program in that folder. The output will look like this:

```
Removing header from NAICS_data_1048.csv...
Removing header from NAICS_data_1218.csv...
```

```
--snip--
Removing header from NAICS_data_9834.csv...
Removing header from NAICS_data_9986.csv...
```

This program should print a filename each time it strips the first line from a CSV file.

Ideas for Similar Programs

Programs that work with CSV files are similar to those that work with Excel files, as CSV and Excel are both spreadsheet files. For example, you could write programs to do the following:

- Compare data between different rows in a CSV file, or between multiple CSV files.
- Copy specific data from a CSV file to an Excel file, or vice versa.
- Check for invalid data or formatting mistakes in CSV files and alert the user about these errors.
- Read data from a CSV file as input for your Python programs.

Versatile Plaintext Formats

While CSV files are useful for storing rows of data that have the exact same columns, the JSON and XML formats can store a variety of data structures. (This book skips the less popular but still useful YAML and TOML formats.) These formats aren't specific to Python; many programming languages have functions for reading and writing data in these formats.

Each of these formats organizes data using the equivalent of nested Python dictionaries and lists. In other programming languages, you'll see dictionaries referred to as *mappings*, *hash maps*, *hash tables*, or *associative arrays* (because they map, or associate, one piece of data, the key, to another, the value). Likewise, you may see Python's lists called *arrays* in other languages. But the concepts are the same: they organize data into key-value pairs and lists.

You can nest dictionaries and lists within other dictionaries and lists to form elaborate data structures. But if you want to save these data structures to a text file, you'll need to choose a data serialization format such as JSON or XML. The Python modules in this chapter can *parse* (that is, read and understand) text written in these formats to create Python data structures from their text.

These human-readable plaintext formats don't make the most efficient use of disk space or memory, but they have the advantage of being easy to view and edit in a text editor and are language neutral, as programs written in any language can read or write text files. By contrast, the `shelve` module, covered in Chapter 10, can store all Python data types in binary shelf files, but other languages don't have modules to load this data into their programs.

In the remainder of this chapter, I'll represent the following Python data structure, which stores personal details about someone named Alice, in each of these formats, so you can compare and contrast them:

```
{
    "name": "Alice Doe",
    "age": 30,
    "car": None,
    "programmer": True,
    "address": {
        "street": "100 Larkin St.",
        "city": "San Francisco",
        "zip": "94102"
    },
    "phone": [
        {
            "type": "mobile",
            "number": "415-555-7890"
        },
        {
            "type": "work",
            "number": "415-555-1234"
        }
    ]
}
```

These text formats have their own histories and occupy specific niches in the computing ecosystem. If you have to choose a data serialization format for storing your data, keep in mind that JSON is simpler than XML and more widely adopted than YAML, and that TOML is chiefly used as a format for configuration files. Lastly, coming up with your own data serialization format might be tempting, but it's also reinventing the wheel, and you would have to write your own parser for your custom format. It's better to simply choose an existing format.

JSON

JSON stores information as JavaScript source code, though many non-JavaScript applications use it. In particular, websites often make their data available to programmers in the JSON format through APIs like the OpenWeather API covered in Chapter 13. We save JSON-formatted text in plaintext files with the *.json* file extension. Here is the example data structure formatted as JSON text:

```
{
  "name": "Alice Doe",
  "age": 30,
  "car": null,
  "programmer": true,
  "address": {
    "street": "100 Larkin St.",
    "city": "San Francisco",
    "zip": "94102"
  },
  "phone": [
    {
      "type": "mobile",
      "number": "415-555-7890"
    },
    {
      "type": "work",
      "number": "415-555-1234"
    }
  ]
}
```

The first thing you'll notice is that JSON is similar to Python syntax. Python's dictionaries and JSON's objects both use curly brackets and contain key-value pairs separated by commas, with each key and value separated by a colon. Python's lists and JSON's arrays both use square brackets and contain values separated by commas. In JSON, whitespace is insignificant outside of double-quoted strings, meaning you can space values however you like. However, it's best to format nested objects and arrays with increased indentation, like blocks of indented Python code. In our example data, the list of phone numbers is indented by two spaces, with each phone number dictionary in the list indented by four spaces.

But there are also differences between JSON and Python. Instead of Python's `None` value, JSON uses the JavaScript keyword `null`. The Boolean values are JavaScript's lowercase `true` and `false` keywords. JSON doesn't allow JavaScript comments or multiline strings; all strings in JSON must use double quotes. Unlike Python lists, JSON arrays can't have trailing commas, so while `["spam", "eggs"]` is valid JSON, `["spam", "eggs",]` is not.

Facebook, Twitter, Yahoo!, Google, Tumblr, Wikipedia, Flickr, Data.gov, Reddit, IMDb, Rotten Tomatoes, LinkedIn, and many other popular sites offer APIs that work with JSON data. Some of these sites require registration, which is almost always free. You'll have to find documentation to learn what URLs your program needs to request in order to get the data you want, as well as the general format of the JSON data structures returned. If the site offering the API has a Developers page, look for the documentation there.

Python's `json` module handles the details of translating between a string formatted as JSON data and corresponding Python values with the `json.loads()` and `json.dumps()` functions. JSON can't store every kind of Python value, only those of the following basic data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`. JSON can't represent Python-specific objects, such as `File` objects, CSV `reader` or `writer` objects, or Selenium `WebElement` objects. The full documentation for the `json` module is at *https://docs.python.org/3/library/json.html*.

Reading JSON Data

To translate a string containing JSON data into a Python value, pass it to the `json.loads()` function. (The name means “load string,” not “loads.”) Enter the following into the interactive shell:

```
❶ >>> import json
>>> json_string = '{"name": "Alice Doe", "age": 30, "car": null, "programmer":
 true, "address": {"street": "100 Larkin St.", "city": "San Francisco", "zip":
 "94102"}, "phone": [{"type": "mobile", "number": "415-555-7890"}, {"type":
"work", "number": "415-555-1234"}]}'
❷ >>> python_data = json.loads(json_string)
>>> python_data
{'name': 'Alice Doe', 'age': 30, 'car': None, 'programmer': True, 'address':
{'street': '100 Larkin St.', 'city': 'San Francisco', 'zip': '94102'},
'phone': [{'type': 'mobile', 'number': '415-555-7890'}, {'type': 'work',
'number': '415-555-1234'}]}
```

After you import the `json` module ❶, you can call `loads()` ❷ and pass it a string of JSON data. Note that JSON strings always use double quotes. It should return the data as a Python dictionary.

Writing JSON Data

The `json.dumps()` function (which means “dump string,” not “dumps”) will translate Python data into a string of JSON-formatted data. Enter the following into the interactive shell:

```
>>> import json
>>> python_data = {'name': 'Alice Doe', 'age': 30, 'car': None, 'programmer': True, 'address':
{'street': '100 Larkin St.', 'city': 'San Francisco', 'zip': '94102'}, 'phone': [{'type':
'mobile', 'number': '415-555-7890'}, {'type': 'work', 'number': '415-555-1234'}]}
>>> json_string = json.dumps(python_data)  ❶
>>> print(json_string) ❷
{"name": "Alice Doe", "age": 30, "car": null, "programmer": true, "address": {"street":
"100 Larkin St.", "city": "San Francisco", "zip": "94102"}, "phone": [{"type": "mobile",
"number": "415-555-7890"}, {"type": "work", "number": "415-555-1234"}]}
>>> json_string = json.dumps(python_data, indent=2) ❸
>>> print(json_string)
{
  "name": "Alice Doe",
  "age": 30,
  "car": null,
  "programmer": true,
  "address": {
    "street": "100 Larkin St.",
    "city": "San Francisco",
--snip--
}
```

The value passed to `json.dumps()` ❶ can consist only of the following basic Python data types: strings, integers, floats, Booleans, lists, dictionaries, and `NoneType`.

By default, the entire JSON text is written on a single line ❷. This compressed format is fine for reading and writing JSON text between programs, but a multiline, indented form would be nicer for humans to read. The `indent=2` keyword argument ❸ formats the JSON text into separate lines, with two spaces of indentation for each nested dictionary or list. Unless your JSON is megabytes in size, increasing the size by adding the space and newline characters is worth it for the readability.

Once you have the JSON text as a Python string value, you can write it to a *.json* file, pass it to a function, use it in a web request, or perform any other operation you can do with a string.

XML

The XML file format is older than JSON but still widely used. Its syntax is similar to HTML, which we covered in Chapter 18, and involves nesting opening and closing tags inside angle brackets that contain other content. These tags are called *elements*. SVG image files are made up of text written in XML. The RSS and Atom web feed formats are also written in XML, and Microsoft Word documents are just ZIP files that have the *.docx* file extension and contain XML files.

We store XML-formatted text in plaintext files with the *.xml* file extension. Here's the example data structure formatted as XML:

```
<person>
    <name>Alice Doe</name>
    <age>30</age>
    <programmer>true</programmer>
    <car xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
    <address>
        <street>100 Larkin St.</street>
        <city>San Francisco</city>
        <zip>94102</zip>
    </address>
    <phone>
        <phoneEntry>
            <type>mobile</type>
            <number>415-555-7890</number>
        </phoneEntry>
        <phoneEntry>
            <type>work</type>
            <number>415-555-1234</number>
        </phoneEntry>
    </phone>
</person>
```

In this example, the `<person>` element has subelements `<name>`, `<age>`, and so on. The `<name>` and `<age>` subelements are *child elements*, and `<person>` is their *parent element*. Valid XML documents must have a single *root element* that contains all the other elements, such as the `<person>` element in this example. A document with multiple root elements like the following is not valid:

```
<person><name>Alice Doe</name></person>
<person><name>Bob Smith</name></person>
<person><name>Carol Watanabe</name></person>
```

XML is quite verbose compared to more modern serialization formats like JSON. Each element has an opening and closing tag, such as `<age>` and `</age>`. An XML element is a key-value pair, with the key being the element's tag (in this case, `<age>`) and the value being the text in between the opening and closing tags. XML text has no data type; everything in between the opening and closing tags is considered a string, including the `94102` and `true` text in our example data. Lists of data, such as the `<phone>` element, have to name their individual items with their own elements, such as `<phoneEntry>`. The "Entry" suffix for these subelements is just a naming convention.

XML's comments are identical to HTML's comments: anything in between `<!--` and `-->` is meant to be ignored.

Whitespace outside the opening and closing tags is insignificant, and you can format it however you like. There is no "null" value in XML, but you can approximate it by adding the `xsi:nil="true"` and `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` attributes to a tag. XML attributes are key-value pairs written in a `key="value"` format within the opening tag. The tag is written as a *self-closing tag*; instead of using a closing tag, the opening tag ends with `/>`, as in `<car xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>`.

Tag and attribute names can be written in any case, but are lowercase by convention. Attribute values can be enclosed in single or double quotes, but double quotes are standard.

Whether to use subelements or attributes is often ambiguous. Our example data uses these elements for the address data:

```
<address>
    <street>100 Larkin St.</street>
    <city>San Francisco</city>
    <zip>94102</zip>
</address>
```

However, it could have easily formatted the subelement data as attributes in a self-closing `<address>` element:

```
<address street="100 Larkin St." city="San Francisco" zip="94102"/>
```

These sorts of ambiguities, as well as the verbose nature of tags, have made XML less popular than it once was. XML was widely deployed throughout the 1990s and 2000s, and much of that software is still used today. But unless you have a specific reason to use XML, you're better served by using JSON.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

In general, XML software libraries have two ways of reading XML documents. The *Document Object Model (DOM)* approach reads the entire XML document into memory at once. This makes it easy to access data anywhere in the XML document, but generally only works for small or moderately sized XML documents. The *Simple API for XML (SAX)* approach reads the XML document as a stream of elements, so it doesn't have to load the entire document into memory at once. This approach is ideal for XML documents that are GBs in size but is less convenient, as you can't work with elements until you've iterated over them in the document.

Python's standard library has the `xml.dom`, `xml.sax`, and `xml.etree.ElementTree` modules for handling XML text. For our simple examples, we'll use Python's `xml.etree.ElementTree` module to read the entire XML document at once.

Reading XML Files

The `xml.etree` module uses `Element` objects to represent an XML element and its child elements. Enter the following into the interactive shell:

```
❶ >>> import xml.etree.ElementTree as ET
❷ >>> xml_string = """<person><name>Alice Doe</name><age>30</age>
<programmer>true</programmer><car xsi:nil="true" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"/><address><street>
100 Larkin St.</street><city>San Francisco</city><zip>94102</zip>
</address><phone><phoneEntry><type>mobile</type><number>415-555-
7890</number></phoneEntry><phoneEntry><type>work</type><number>
415-555-1234</number></phoneEntry></phone></person>"""
❸ >>> root = ET.fromstring(xml_string)
>>> root
<Element 'person' at 0x000001942999BBA0>
```

We import the `xml.etree.ElementTree` module ❶ with the `as ET` syntax so that we can enter `ET` instead of the long `xml.etree.ElementTree` module name. The `xml_string` variable ❷ contains the text of the XML we wish to parse, though this text could have just as easily been read from a text file with the *.xml* file extension. Finally, we pass this text to the `ET.fromstring()` function ❸, which returns an `Element` object containing the data we want to access. We'll store this `Element` object in a variable named `root`.

The `xml.etree.ElementTree` module also has a `parse()` function. You can pass it the name of a file from which to load XML, and it returns an `Element` object:

```
>>> import xml.etree.ElementTree as ET
>>> tree = ET.parse('my_data.xml')
>>> root = tree.getroot()
```

Once you have an `Element` object, you can access its `tag` and `text` Python attributes to see the name of the tag, as well as the text enclosed within its opening and closing tags. If you pass the `Element` object to the `list()`

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

function, it should return a list of its immediate child elements. Continue the interactive shell by entering the following:

```
>>> root.tag
'person'
>>> list(root)
[<Element 'name' at 0x00000150BA4ADDF0>, <Element 'age' at
0x00000150BA4ADF30>, <Element 'programmer' at 0x00000150BA4ADEE0>,
<Element 'car' at 0x00000150BA4ADD00>, <Element 'address' at
0x00000150BA4ADCB0>, <Element 'phone' at 0x00000150BA4ADA30>]
```

The child Element objects of a parent Element object are accessible through an integer index, just like Python lists. So, if root contains the <person> element, then root[0] and root[1] contain the <name> and <age> elements, respectively. You can access the tag and text attributes of all of these Element objects. However, any self-closing tags, like <car/>, will use None for their text attribute. For example, enter the following into the interactive shell:

```
>>> root[0].tag
'name'
>>> root[0].text
'Alice Doe'
>>> root[3].tag
'car'
>>> root[3].text == None  # <car/> has no text.
True
>>> root[4].tag
'address'
>>> root[4][0].tag
'street'
>>> root[4][0].text
'100 Larkin St.'
```

From the root element, you can explore the data in the entire XML document. You can also iterate over the immediate child elements by putting an Element object in a for loop:

```
>>> for elem in root:
...   print(elem.tag, '--', elem.text)
...
name -- Alice Doe
age -- 30
programmer -- true
car -- None
address -- None
phone -- None
```

If you want to iterate over all children underneath the Element, you can call the iter() method in a for loop:

```
>>> for elem in root.iter():
...   print(elem.tag, '--', elem.text)
```

```
...
person -- None
name -- Alice Doe
age -- 30
programmer -- true
car -- None
address -- None
street -- 100 Larkin St.
city -- San Francisco
zip -- 94102
phone -- None
phoneEntry -- None
type -- mobile
number -- 415-555-7890
phoneEntry -- None
type -- work
number -- 415-555-1234
```

Optionally, you can pass a string to the `iter()` method to filter for XML elements with a matching tag. This example calls `iter('number')` to iterate over only the `<number>` child elements of the root element:

```
>>> for elem in root.iter('number'):
...   print(elem.tag, '--', elem.text)
...
number -- 415-555-7890
number -- 415-555-1234
```

There's much more to browsing the data in an XML document than the attributes and methods covered in this section. For example, just as the CSS selectors covered in Chapter 13 can find elements in a web page's HTML, a language called *XPath* can locate elements in an XML document. These concepts are beyond the scope of this chapter, but you can learn about them in the Python documentation at *https://docs.python.org/3/library/xml.etree.elementtree.html*.

Python's XML modules have no way to convert XML text to a Python data structure. However, the third-party `xmltodict` module at *https://pypi.org/project/xmltodict/* can do this. The full installation instructions are in Appendix A. Here is an example of its use:

```
>>> import xmltodict
>>> xml_string = """<person><name>Alice Doe</name><age>30</age>
<programmer>true</programmer><car xsi:nil="true" xmlns:xsi=
"http://www.w3.org/2001/XMLSchema-instance"/><address><street>
100 Larkin St.</street><city>San Francisco</city><zip>94102
</zip></address><phone><phoneEntry><type>mobile</type><number>
415-555-7890</number></phoneEntry><phoneEntry><type>work</type>
<number>415-555-1234</number></phoneEntry></phone></person>"""
>>> python_data = xmltodict.parse(xml_string)
>>> python_data
{'person': {'name': 'Alice Doe', 'age': '30', 'programmer': 'true',
'car': {'@xsi:nil': 'true', '@xmlns:xsi': 'http://www.w3.org/2001/
XMLSchema-instance'}, 'address': {'street': '100 Larkin St.', 'city':
'San Francisco', 'zip': '94102'}, 'phone': {'phoneEntry': [{'type':
'mobile', 'number': '415-555-7890'}, {'type': 'work', 'number':
'415-555-1234'}]}}}
```

One reason the XML standard has fallen to the wayside compared to formats like JSON is that representing data types in XML is more complicated. For example, the `<programmer>` element was parsed as the string value `'true'` instead of the Boolean value `True`. And the `<car>` element was parsed into the awkward `'car': {'@xsi:nil': 'true', '@xmlns:xsi': 'http://www.w3.org/2001/XMLSchema-instance'}` key-value pair instead of the value `None`. You must double-check the input and output of any XML module to verify that it is representing your data as you intend.

Writing XML Files

The `xml.etree` module is a bit unwieldy, so for small projects, you may be better off calling the `open()` function and `write()` method to create XML text yourself. But to create an XML document from scratch with the `xml.etree` module, you'll need to create a root `Element` object (such as the `<person>` element in our example) and then call the `SubElement()` function to create child elements for it. You can set any XML attributes in the element with the `set()` method. For example, enter the following:

```
>>> import xml.etree.ElementTree as ET
>>> person = ET.Element('person')  # Create the root XML element.
>>> name = ET.SubElement(person, 'name')  # Create <name> and put it under <person>.
>>> name.text = 'Alice Doe'  # Set the text between <name> and </name>.
>>> age = ET.SubElement(person, 'age')
>>> age.text = '30'  # XML content is always a string.
>>> programmer = ET.SubElement(person, 'programmer')
>>> programmer.text = 'true'
>>> car = ET.SubElement(person, 'car')
>>> car.set('xsi:nil', 'true')
>>> car.set('xmlns:xsi', 'http://www.w3.org/2001/XMLSchema-instance')
>>> address = ET.SubElement(person, 'address')
>>> street = ET.SubElement(address, 'street')
>>> street.text = '100 Larkin St.'
```

For brevity, we'll leave out the rest of the `<address>` and `<phone>` elements. Call the `ET.tostring()` and `decode()` functions with the root `Element` object to get a Python string of the XML text:

```
>>> ET.tostring(person, encoding='utf-8').decode('utf-8')
'<person><name>Alice Doe</name><age>30</age><programmer>true</programmer>
<car xsi:nil="true" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"/>
<address><street>100 Larkin St.</street></address></person>'
```

It's rather unfortunate that the `tostring()` method returns a `bytes` object instead of a string, necessitating a `decode()` method call to obtain an actual string. But once you have the XML text as a Python string value, you can write it to a *.xml* file, pass it to a function, use it in a web request, or do anything else you can do with a string.

Summary

CSV, JSON, and XML are common plaintext formats for storing data. They're easy for programs to parse while still being human readable, so they are often used for simple spreadsheets or web app data. The `csv`, `json`, and `xml.etree.ElementTree` modules in the Python standard library greatly simplify the process of reading and writing these files, so you don't need to do so with the `open()` function.

These formats are not specific to Python; many other programming languages and software applications use these file types. This chapter can help you write Python programs that can also interact with any apps that use them.

Practice Questions

1. What are some features that Excel spreadsheets have but CSV spreadsheets don't?
2. What do you pass to `csv.reader()` and `csv.writer()` to create `reader` and `writer` objects?
3. What modes do `File` objects for `reader` and `writer` objects need to be opened in?
4. What method takes a list argument and writes it to a CSV file?
5. What do the `delimiter` and `lineterminator` keyword arguments do?
6. Of CSV, JSON, and XML, which formats can be easily edited with a text editor application?
7. What function takes a string of JSON data and returns a Python data structure?
8. What function takes a Python data structure and returns a string of JSON data?
9. Which data serialization format resembles HTML, with tags enclosed in angle brackets?
10. How does JSON write `None` values?
11. How do you write Boolean values in JSON?

Practice Program: Excel-to-CSV Converter

Excel can save a spreadsheet to a CSV file with a few mouse clicks, but if you had to convert hundreds of Excel files to CSVs, it would take hours of clicking. Using the `openpyxl` module from Chapter 14, write a program that reads all the Excel files in the current working directory and outputs them as CSV files.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

A single Excel file might contain multiple sheets; you'll have to create one CSV file per sheet. The filenames of the CSV files should be *<excel filename>_<sheet title>.csv*, where *<excel filename>* is the filename of the Excel file without the file extension (for example, *spam_data*, not *spam_data.xlsx*) and *<sheet title>* is the string from the `Worksheet` object's `title` variable.

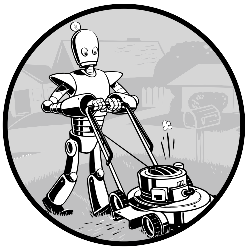
This program will involve many nested `for` loops. The skeleton of the program should look something like this:

```
for excel_file in os.listdir('.'):
    # Skip non-xlsx files, load the workbook object.
    for sheet_name in wb.sheetnames:
        # Loop through every sheet in the workbook.
        # Create the CSV filename from the Excel filename and sheet title.
        # Create the csv.writer object for this CSV file.

        # Loop through every row in the sheet.
        for row_num in range(1, sheet.max_row + 1):
            row_data = []    # append each cell to this list
            # Loop through each cell in the row.
            for col_num in range(1, sheet.max_column + 1):
                # Append each cell's data to row_data

            # Write the row_data list to the CSV file.

        csv_file.close()
```

Download the ZIP file *excelSpreadsheets.zip* from the book's online resources and unzip the spreadsheets into the same directory as your program. You can use these as the files to test the program on.

19

KEEPING TIME, SCHEDULING TASKS, AND LAUNCHING PROGRAMS



Running programs while you're sitting at your computer is fine, but it's also useful to have programs run without your direct supervision. Your computer's clock can schedule programs to run code at some specified time and date or at regular intervals. For example, your program could scrape a website every hour to check for changes or do a CPU-intensive task at 4 AM while you sleep. Python's `time` and `datetime` modules provide these functions.

You can also write programs that launch other programs on a schedule by using the `subprocess` and `threading` modules. Often, the fastest way to program is to take advantage of applications that other people have already written.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The time Module

Your computer's system clock is set to a specific date, time, and time zone. The built-in `time` module allows your Python programs to read the system clock for the current time. The most useful of its functions are `time.time()`, which returns a value called the epoch timestamp, and `time.sleep()`, which pauses a program.

Returning the Epoch Timestamp

The *Unix epoch* is a time reference commonly used in programming: midnight on January 1, 1970, Coordinated Universal Time (UTC). The `time.time()` function returns the number of seconds since that moment as a float value. (Recall that a float is just a number with a decimal point.) This number is called an *epoch timestamp*. For example, enter the following into the interactive shell:

```
>>> import time
>>> time.time()
1773813875.3518236
```

Here, I'm calling `time.time()` on March 17, 2026, at 11:04 PM Pacific Standard Time. The return value is how many seconds have passed between the Unix epoch and the moment `time.time()` was called.

The return value from `time.time()` is useful, but is not human readable. The `time.ctime()` function returns a string description of the current time. You can also optionally pass the number of seconds since the Unix epoch, as returned by `time.time()`, to get a string value of that time. Enter the following into the interactive shell:

```
>>> import time
>>> time.ctime()
'Tue Mar 17 11:05:38 2026'
>>> this_moment = time.time()
>>> time.ctime(this_moment)
'Tue Mar 17 11:05:45 2026'
```

Epoch timestamps can be used to *profile* code: that is, measure how long a piece of code takes to run. If you call `time.time()` at the beginning of the code block you want to measure and again at the end, you can subtract the first timestamp from the second to find the elapsed time between those two calls. For example, open a new file editor tab and enter the following program:

```
# Measure how long it takes to multiply 100,000 numbers.
import time
❶ def calculate_product():
    # Calculate the product of the first 100,000 numbers.
    product = 1
    for i in range(1, 100001):
        product = product * i
    return product
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
❷ start_time = time.time()
  result = calculate_product()
❸ end_time = time.time()
❹ print(f'It took {end_time - start_time} seconds to calculate.')
```

At ❶, we define a function `calculate_product()` to loop through the integers from 1 to 100,000 and return their product. At ❷, we call `time.time()` and store it in `start_time`. Right after calling `calculate_product()`, we call `time.time()` again and store it in `end_time` ❸. We end by printing how long it took to run `calculate_product()` ❹.

Save this program as *calcProd.py* and run it. The output will look something like this:

```
It took 2.844162940979004 seconds to calculate.
```

Another way to profile your code is to use the `cProfile.run()` function, which provides a much more informative level of detail than the simple `time.time()` technique. You can read about `cProfile.run()` function in Chapter 13 of my other book, *Beyond the Basic Stuff with Python* (No Starch Press, 2020).

Pausing Programs

If you need to pause your program for a while, call the `time.sleep()` function and pass it the number of seconds you want your program to stay paused. For example, enter the following into the interactive shell:

```
  >>> import time
  >>> for i in range(3):
  ...   ❶ print('Tick')
  ...   ❷ time.sleep(1)
  ...   ❸ print('Tock')
  ...   ❹ time.sleep(1)
  ...
  Tick
  Tock
  Tick
  Tock
  Tick
  Tock
❺ >>> time.sleep(5)
  >>>
```

The `for` loop will print `Tick` ❶, pause for one second ❷, print `Tock` ❸, pause for one second ❹, print `Tick`, pause, and so on, until `Tick` and `Tock` have each been printed three times.

The `time.sleep()` function will *block* (that is, it won't return or release your program to execute other code) until after the number of seconds you passed to `time.sleep()` has elapsed. For example, if you enter `time.sleep(5)` ❺, you'll see that the next prompt (`>>>`) doesn't appear until five seconds have passed.

Rounding Numbers

When working with time, you'll often encounter float values with many digits after the decimal. To make these values easier to work with, you can shorten them with Python's built-in `round()` function, which rounds a float to the precision you specify. Just pass in the number you want to round, plus an optional second argument representing how many digits after the decimal point you want to round it to. If you omit the second argument, `round()` rounds your number to the nearest whole integer. Enter the following into the interactive shell:

```
>>> import time
>>> now = time.time()
>>> now
1773814036.6147408
>>> round(now, 2)
1773814036.61
>>> round(now, 4)
1773814036.6147
>>> round(now)
1773814037
```

After importing `time` and storing `time.time()` in `now`, we call `round(now, 2)` to round `now` to two digits after the decimal, `round(now, 4)` to round to four digits after the decimal, and `round(now)` to round to the nearest integer.

Project 14: Super Stopwatch

Say you want to track how much time you spend on boring tasks you haven't automated yet. You don't have a physical stopwatch, and it's surprisingly difficult to find a free stopwatch app for your laptop or smartphone that isn't covered in ads and doesn't send a copy of your browser history to marketers. (It says it can do this in the license agreement you agreed to. You did read the license agreement, didn't you?) You can write a simple stopwatch program yourself in Python.

At a high level, here's what your program will do:

- Find the current time by calling `time.time()` and store it as a timestamp at the start of the program, as well as at the start of each lap.
- Keep a lap counter and increment it every time the user presses ENTER.
- Calculate the elapsed time by subtracting timestamps.
- Handle the `KeyboardInterrupt` exception so that the user can press CTRL-C to quit.

Open a new file editor tab and save it as *stopwatch.py*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Step 1: Set Up the Program to Track Times

The stopwatch program will need to use the current time, so you'll want to import the `time` module. Your program should also print some brief instructions to the user before calling `input()` so that the timer can begin after the user presses ENTER. Then, the code will start tracking lap times each time the user presses ENTER until they press CTRL-C to quit.

Enter the following code into the file editor, writing a `TODO` comment as a placeholder for the rest of the code:

```
# A simple stopwatch program
import time

# Display the program's instructions.
print('Press ENTER to begin and to mark laps. Ctrl-C quits.')
input()  # Press Enter to begin.
print('Started.')
start_time = time.time()  # Get the first lap's start time.
last_time = start_time
lap_number = 1

# TODO: Start tracking the lap times.
```

Now that you've written the code to display the instructions, start the first lap, note the time, and set the `lap_number` to 1.

Step 2: Track and Print Lap Times

Now let's write the code to start each new lap, calculate how long the previous lap took, and calculate the total time elapsed since starting the stopwatch. We'll display the lap time and total time and increase the lap count for each new lap. Add the following code to your program:

```
# A simple stopwatch program
import time

--snip--

# Start tracking the lap times.
❶ try:
    ❷ while True:
        input()
        ❸ lap_time = round(time.time() - last_time, 2)
        ❹ total_time = round(time.time() - start_time, 2)
        ❺ print('Lap #{lap_number}: {total_time}({lap_time})', end='')
        lap_number += 1
        last_time = time.time() # Reset the last lap time.
❻ except KeyboardInterrupt:
    # Handle the Ctrl-C exception to keep its error message from displaying.
    print('\nDone.')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If the user presses CTRL-C to stop the stopwatch, the KeyboardInterrupt exception will be raised, and the program will crash. To prevent crashing, we wrap this part of the program in a try statement ❶. We'll handle the exception in the except clause ❻, which prints Done when the exception is raised instead of showing the KeyboardInterrupt error message. Until this happens, the execution occurs inside an infinite loop ❷ that calls input() and waits until the user presses ENTER to end a lap. When a lap ends, we calculate how long the lap took by subtracting the start time of the lap, last_time, from the current time, time.time() ❸. We calculate the total time elapsed by subtracting the overall start time of the stopwatch, start_time, from the current time ❹.

Because the results of these time calculations will have many digits after the decimal point (such as 4.766272783279419), we use the round() function to round the float value to two digits at ❸ and ❹.

At ❺, we print the lap number, total time elapsed, and lap time. As the user pressing ENTER for the input() call will print a newline to the screen, pass end='' to the print() function to avoid double-spacing the output. After printing the lap information, we get ready for the next lap by adding 1 to the count lap_number and setting last_time to the current time, which is the start time of the next lap.

Ideas for Similar Programs

Time tracking opens up several possibilities for your programs. Although you can download apps to do some of these things, the benefit of writing programs yourself is that they will be free and not bloated with ads and useless features. You could write similar programs to do the following:

- Create a simple timesheet app that records when you type a person's name and uses the current time to clock them in or out.
- Add a feature to your program to display the elapsed time since a process started, such as a download that uses the requests module. (See Chapter 13.)
- Intermittently check how long a program has been running and offer the user a chance to cancel tasks that are taking too long.

The datetime Module

The time module is useful for getting a Unix epoch timestamp to work with. But if you want to display a date in a more convenient format, or do arithmetic with dates (for example, figuring out what date was 205 days ago or what date is 123 days from now), you should use the datetime module.

The datetime module has its own datetime data type. The datetime values represent a specific moment in time. Enter the following into the interactive shell:

```
  >>> import datetime
❶ >>> datetime.datetime.now()
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
❷ datetime.datetime(2026, 2, 27, 11, 10, 49, 727297)
❸ >>> dt = datetime.datetime(2026, 10, 21, 16, 29, 0)
❹ >>> dt.year, dt.month, dt.day
  (2026, 10, 21)
❺ >>> dt.hour, dt.minute, dt.second
  (16, 29, 0)
```

Calling `datetime.datetime.now()` ❶ returns a `datetime` object ❷ for the current date and time, according to your computer's clock. This object includes the year, month, day, hour, minute, second, and microsecond of the current moment. You can also retrieve a `datetime` object for a specific moment by using the `datetime.datetime()` function ❸, passing it integers representing the year, month, day, hour, and second of the moment you want. These integers will be stored in the `datetime` object's `year`, `month`, `day` ❹, `hour`, `minute`, and `second` ❺ attributes.

A Unix epoch timestamp can be converted to a `datetime` object with the `datetime.datetime.fromtimestamp()` function. The date and time of the `datetime` object will be converted for the local time zone. Enter the following into the interactive shell:

```
>>> import datetime, time
>>> datetime.datetime.fromtimestamp(1000000)
datetime.datetime(1970, 1, 12, 5, 46, 40)
>>> datetime.datetime.fromtimestamp(time.time())
datetime.datetime(2026, 10, 21, 16, 30, 0, 604980)
```

Calling `datetime.datetime.fromtimestamp()` and passing it `1000000` returns a `datetime` object for the moment 1,000,000 seconds after the Unix epoch. Passing `time.time()`, the Unix epoch timestamp for the current moment, returns a `datetime` object for the current moment. So, the expressions `datetime.datetime.now()` and `datetime.datetime.fromtimestamp(time.time())` do the same thing; they both give you a `datetime` object for the present moment.

You can compare `datetime` objects with each other using comparison operators to find out which one precedes the other. The later `datetime` object is the "greater" value. Enter the following into the interactive shell:

```
  >>> import datetime
❶ >>> halloween_2026 = datetime.datetime(2026, 10, 31, 0, 0, 0)
❷ >>> new_years_2027 = datetime.datetime(2027, 1, 1, 0, 0, 0)
  >>> oct_31_2026 = datetime.datetime(2026, 10, 31, 0, 0, 0)
❸ >>> halloween_2026 == oct_31_2026
  True
❹ >>> halloween_2026 > new_years_2027
  False
❺ >>> new_years_2027 > halloween_2026
  True
  >>> new_years_2027 != oct_31_2026
  True
```

This code makes a `datetime` object for the first moment (midnight) of October 31, 2026, and stores it in `halloween_2026` ❶. Then, it makes a `datetime` object for the first moment of January 1, 2027, and stores it in new_years_2027 ❷. It creates another object for midnight on October 31, 2026, and stores it in oct_31_2026. Comparing halloween_2026 and oct_31_2026 shows that they're equal ❸. Comparing new_years_2027 and halloween_2026 shows that new_years_2027 is greater (later) than halloween_2026 ❹ ❺.

Representing Duration

The datetime module also provides a timedelta data type, which represents a *duration* of time rather than a *moment* in time. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> delta = datetime.timedelta(days=11, hours=10, minutes=9, seconds=8)
❷ >>> delta.days, delta.seconds, delta.microseconds
(11, 36548, 0)
>>> delta.total_seconds()
986948.0
>>> str(delta)
'11 days, 10:09:08'
```

To create a timedelta object, use the datetime.timedelta() function. The datetime.timedelta() function takes the keyword arguments weeks, days, hours, minutes, seconds, milliseconds, and microseconds. There is no month or year keyword argument, because "a month" or "a year" is a variable amount of time depending on the particular month or year. A timedelta object has the total duration represented in days, seconds, and microseconds. These numbers are stored in the days, seconds, and microseconds attributes, respectively. The total_seconds() method will return the duration in number of seconds alone. Passing a timedelta object to str() will return a nicely formatted, human-readable string representation of the object.

In this example, we pass keyword arguments to datetime.delta() to specify a duration of 11 days, 10 hours, 9 minutes, and 8 seconds, and store the returned timedelta object in delta ❶. This timedelta object's days attribute stores 11, and its seconds attribute stores 36548 (10 hours, 9 minutes, and 8 seconds, expressed in seconds) ❷. Calling total_seconds() tells us that 11 days, 10 hours, 9 minutes, and 8 seconds is 986,948 seconds. Finally, passing the timedelta object to str() returns a string that plainly describes the duration.

The arithmetic operators can be used to perform *date arithmetic* on datetime values. For example, to calculate the date 1,000 days from now, enter the following into the interactive shell:

```
>>> import datetime
>>> now = datetime.datetime.now()
>>> now
datetime.datetime(2026, 12, 2, 18, 38, 50, 636181)
>>> thousand_days = datetime.timedelta(days=1000)
>>> now + thousand_days
datetime.datetime(2029, 8, 28, 18, 38, 50, 636181)
```

First, make a datetime object for the current moment and store it in now. Then, make a timedelta object for a duration of 1,000 days and store it in thousand_days. Add now and thousand_days together to get a datetime object for the date 1,000 days from the date and time in now. Python will do the date arithmetic to figure out that 1,000 days after December 2, 2026, will be August 28, 2029. When calculating 1,000 days from a given date, you have to remember how many days are in each month and factor in leap years and other tricky details. The datetime module handles all of this for you.

You can add or subtract timedelta objects with datetime objects or other timedelta objects using the + and - operators. A timedelta object can be multiplied or divided by integer or float values with the * and / operators. Enter the following into the interactive shell:

```
>>> import datetime
❶ >>> oct_21st = datetime.datetime(2026, 10, 21, 16, 29, 0)
❷ >>> about_thirty_years = datetime.timedelta(days=365 * 30)
>>> oct_21st
datetime.datetime(2026, 10, 21, 16, 29)
>>> oct_21st - about_thirty_years
datetime.datetime(1996, 10, 28, 16, 29)
>>> oct_21st - (2 * about_thirty_years)
datetime.datetime(1966, 11, 5, 16, 29)
```

Here, we make a datetime object for October 21, 2026, ❶ and a timedelta object for a duration of about 30 years ❷. (We're using 365 days for each of those years and ignoring leap years.) Subtracting about_thirty_years from oct_21st gives us a datetime object for the date 30 years before October 21, 2026. Subtracting 2 * about_thirty_years from oct_21st returns a datetime object for the date about 60 years before: the late afternoon of November 5, 1966.

Pausing Until a Specific Date

The time.sleep() method lets you pause a program for a certain number of seconds. By using a while loop, you can pause your programs until a specific date. For example, the following code will continue to loop until Halloween 2039:

```
import datetime
import time
halloween_2039 = datetime.datetime(2039, 10, 31, 0, 0, 0)
while datetime.datetime.now() < halloween_2039:
    time.sleep(1)  # Wait 1 second before looping to check again.
```

The time.sleep(1) call will pause your Python program so that the computer doesn't waste CPU processing cycles by checking the time over and over as fast as possible. Rather, the while loop will just check the condition once per second and continue with the rest of the program after Halloween 2039 (or whenever you program it to stop).

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Converting datetime Objects into Strings

Epoch timestamps and datetime objects aren't very friendly to the human eye. Use the strftime() method to display a datetime object as a string. (The *f* in the name of the strftime() function stands for *format.*)

The strftime() method uses directives similar to Python's string formatting. Table 19-1 has a full list of strftime() directives. You can also consult the helpful *https://strftime.org* website for this information.

Table 19-1: strftime() Directives

strftime() directive	Meaning
%Y	Year with century, as in '2026'
%y	Year without century, '00' to '99' (1970 to 2069)
%m	Month as a decimal number, '01' to '12'
%B	Full month name, as in 'November'
%b	Abbreviated month name, as in 'Nov'
%d	Day of the month, '01' to '31'
%j	Day of the year, '001' to '366'
%w	Day of the week, '0' (Sunday) to '6' (Saturday)
%A	Full weekday name, as in 'Monday'
%a	Abbreviated weekday name, as in 'Mon'
%H	Hour (24-hour clock), '00' to '23'
%I	Hour (12-hour clock), '01' to '12'
%M	Minute, '00' to '59'
%S	Second, '00' to '59'
%p	'AM' or 'PM'
%%	Literal '%' character

Pass strftime() a custom format string containing formatting directives (along with any desired slashes, colons, and so on), and strftime() will return the datetime object's information as a formatted string. Enter the following into the interactive shell:

```
>>> oct_21st = datetime.datetime(2026, 10, 21, 16, 29, 0)
>>> oct_21st.strftime('%Y/%m/%d %H:%M:%S')
'2026/10/21 16:29:00'
>>> oct_21st.strftime('%I:%M %p')
'04:29 PM'
>>> oct_21st.strftime("%B of '%y")
"October of '26"
```

Here, we have a datetime object for October 21, 2026, at 4:29 PM, stored in oct_21st. Passing the custom format string '%Y/%m/%d %H:%M:%S' to strftime() returns a string containing 2026, 10, and 21 separated by slashes and 16, 29,

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

and 00 separated by colons. Passing `'%I:%M %p'` returns `'04:29 PM'`, and passing `"%B of '%y"` returns `"October of '26"`.

Converting Strings into datetime Objects

If you have a string of date information, such as `'2026/10/21 16:29:00'` or `'October 21, 2026'`, and you need to convert it to a `datetime` object, use the `datetime.datetime.strptime()` function. The `strptime()` function is the inverse of the `strftime()` method, and you must pass it a custom format string using the same directives as `strftime()` so that the function knows how to parse and understand the string. (The *p* in the name of the `strptime()` function stands for *parse*.)

Enter the following into the interactive shell:

```
❶ >>> datetime.datetime.strptime('October 21, 2026', '%B %d, %Y')
datetime.datetime(2026, 10, 21, 0, 0)
>>> datetime.datetime.strptime('2026/10/21 16:29:00', '%Y/%m/%d %H:%M:%S')
datetime.datetime(2026, 10, 21, 16, 29)
>>> datetime.datetime.strptime("October of '26", "%B of '%y")
datetime.datetime(2026, 10, 1, 0, 0)
>>> datetime.datetime.strptime("November of '63", "%B of '%y")
datetime.datetime(2063, 11, 1, 0, 0)
❷ >>> datetime.datetime.strptime("November of '73", "%B of '%y")
datetime.datetime(1973, 11, 1, 0, 0)
```

To get a `datetime` object from the string `'October 21, 2026'`, pass that string as the first argument to `strptime()` and the custom format string that corresponds to `'October 21, 2026'` as the second argument ❶. The string with the date information must match the custom format string exactly, or Python will raise a `ValueError` exception. Notice that `"November of '63"` is interpreted as 2063 while `"November of '73"` is interpreted as 1973 because the `%y` directive spans from 1970 to 2069.

A REVIEW OF PYTHON'S TIME FUNCTIONS

Dates and times in Python can involve quite a few different data types and functions. Here's a review of the three different types of values used to represent time:

- A Unix epoch timestamp (used by the `time` module) is a float or integer value representing the number of seconds since midnight on January 1, 1970, UTC.
- A `datetime` object (of the `datetime` module) has integers stored in the attributes `year`, `month`, `day`, `hour`, `minute`, `second`, and `microsecond`.
- A `timedelta` object (of the `datetime` module) represents a time duration, rather than a specific moment.

(continued)

Here's a review of time functions, their parameters, and their return values:

`time.time()` This function returns an epoch timestamp of the current moment as a float value.

`time.sleep(`*`seconds`*`)` This function stops the program for the number of seconds specified by the *`seconds`* argument.

`datetime.datetime(`*`year, month, day, hour, minute, second`*`)` This function returns a `datetime` object of the moment specified by the arguments. If *`hour`*, *`minute`*, or *`second`* arguments are not provided, they default to `0`.

`datetime.datetime.now()` This function returns a `datetime` object of the current moment.

`datetime.datetime.fromtimestamp(`*`epoch`*`)` This function returns a `datetime` object of the moment represented by the *`epoch`* timestamp argument.

`datetime.timedelta(`*`weeks, days, hours, minutes, seconds, milliseconds, microseconds`*`)` This function returns a `timedelta` object representing a duration of time. The function's keyword arguments are all optional and do not include *`month`* or *`year`*.

`total_seconds()` This `timedelta` method returns the number of seconds the `timedelta` object represents.

`strftime(`*`format`*`)` This `datetime` method returns a string of the time in a custom format based on the *`format`* string. See Table 19-1 for the format details.

`datetime.datetime.strptime(`*`time_string, format`*`)` This function returns a `datetime` object representing the moment specified by *`time_string`*, parsed using the *`format`* string argument. See Table 19-1 for the format details.

Launching Other Programs from Python

Your Python program can start other programs on your computer with the `run()` function in the built-in `subprocess` module. If you have multiple instances of an application open, each of those instances is a separate process of the same program. For example, each open window of the calculator app shown in Figure 19-1 is a different process.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

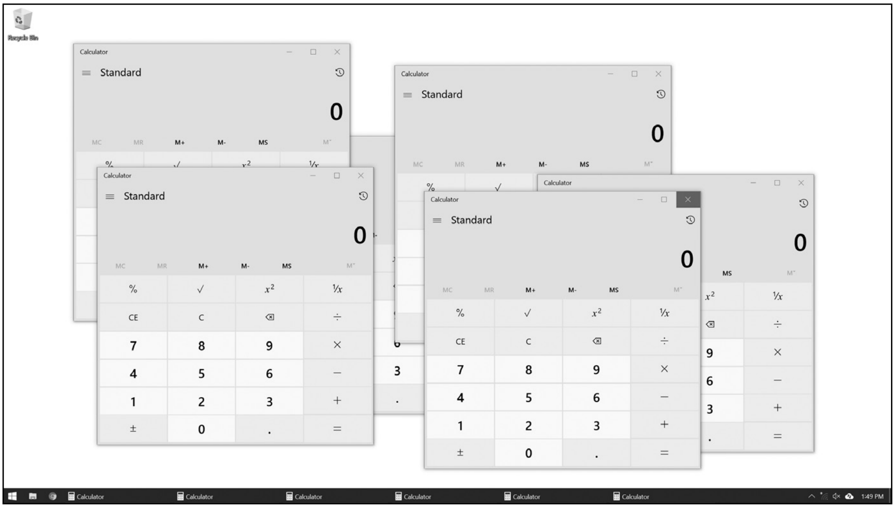


Figure 19-1: Six running processes of the same calculator program

If you want to start an external program from your Python script, pass the program's filename to `subprocess.run()`. (On Windows, right-click the application's **Start** menu item and select **Properties** to view the application's filename. On macOS, CTRL-click the application and select **Show Package Contents** to find the path to the executable file.) The `run()` function will block until the launched program closes. Pass the program to launch as a string of the executable program's filepath inside a list, keeping in mind that the launched program will be run in a separate process, not in the same process as your Python program.

On a Windows computer, enter the following into the interactive shell:

```
>>> import subprocess
>>> subprocess.run(['C:\\Windows\\System32\\calc.exe'])
CompletedProcess(args=['C:\\Windows\\System32\\calc.exe'], returncode=0)
```

On Ubuntu Linux, enter the following:

```
>>> import subprocess
>>> subprocess.run(['/usr/bin/gnome-calculator'])
CompletedProcess(args=['/usr/bin/gnome-calculator'], returncode=0)
```

On macOS, enter the following:

```
>>> import subprocess
>>> subprocess.run(['open', '/System/Applications/Calculator.app'])
CompletedProcess(args=['open', '/System/Applications/Calculator.app'], returncode=0)
```

Notice that macOS requires you to run the `open` program and pass it a command line argument of the program you want to launch.

In these examples, our Python code launched the program, waited for the program to close, and then continued. If you want your Python code to launch a program and then immediately continue without waiting for the program to close, call the `subprocess.Popen()` ("process open") function instead:

```
>>> import subprocess
>>> calc_proc = subprocess.Popen(['C:\\Windows\\System32\\calc.exe'])
```

The return value is a `Popen` object, which has two useful methods: `poll()` and `wait()`.

You can think of the `poll()` method as asking your driver "Are we there yet?" over and over until you arrive. The `poll()` method will return `None` if the process is still running at the time `poll()` is called. If the program has terminated, it will return the process's integer *exit code*. An exit code indicates whether the process terminated without errors (represented by an exit code of `0`) or whether an error caused the process to terminate (represented by a nonzero exit code—generally `1`, but it may vary depending on the program).

The `wait()` method is like waiting until the driver has arrived at your destination. The method will block until the launched process has terminated. This is helpful if you want your program to pause until the user finishes interacting with the other program. The return value of `wait()` is the process's integer exit code.

On Windows, enter the following into the interactive shell. Note that the `wait()` call may block until you quit the launched Calculator program:

```
  >>> import subprocess
❶ >>> calc_proc = subprocess.Popen('c:\\Windows\\System32\\calc.exe')
❷ >>> calc_proc.poll() == None
  True
❸ >>> calc_proc.wait() # Doesn't return until Calculator closes.
  0
  >>> calc_proc.poll()
  0
```

Here, we open a Calculator process ❶. On older versions of Windows, `poll()` returns `None` ❷ if the process is still running. Then, we close the Calculator application's window, and back in the interactive shell, we call `wait()` on the terminated process ❸. Now `wait()` and `poll()` return `0`, indicating that the process terminated without errors.

If you run *calc.exe* on Windows 10 and later using `subprocess.Popen()`, you'll notice that `wait()` instantly returns even though the calculator app is still running. This is because *calc.exe* launches the calculator app and then instantly closes itself. The calculator program in Windows is a "Trusted Microsoft Store app," and its specifics are beyond the scope of this book. Suffice it to say that programs can run in many application-specific and operating system–specific ways.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If you want to close a process you've launched with `subprocess.Popen()`, call the `kill()` method of the `Popen` object the function returned. If you have MS Paint on Windows, enter the following into the interactive shell:

```
>>> import subprocess
>>> paint_proc = subprocess.Popen('c:\\Windows\\System32\\mspaint.exe')
>>> paint_proc.kill()
```

Note that the `kill()` method immediately terminates a program and bypasses any "Are you sure you want to quit?" confirmation window. Any unsaved work in the program will be lost.

Passing Command Line Arguments to Processes

You can pass command line arguments to processes you create with `run()`. To do so, pass a list as the sole argument to `run()`. The first string in this list will be the executable filename of the program you want to launch; all the subsequent strings will be the command line arguments to pass to the program when it starts. In effect, this list will be the value of `sys.argv` for the launched program.

Most applications with a graphical user interface (GUI) don't use command line arguments as extensively as command line–based or terminal-based programs do. But most GUI applications will accept a single argument for a file that the applications will immediately open when they start. For example, if you're using Windows, create a text file named *C:\Users\Al\hello.txt* and then enter the following into the interactive shell:

```
>>> subprocess.run(['C:\\Windows\\notepad.exe', 'C:\\Users\Al\\hello.txt'])
CompletedProcess(args=['C:\\Windows\\notepad.exe', 'C:\\Users\\Al\\hello.txt'], returncode=0)
```

This will not only launch the Notepad application but also have it immediately open the *C:\Users\Al\hello.txt* file. Every program has its own set of command line arguments, and some programs (especially GUI applications) don't use command line arguments at all.

The `subprocess.Popen()` function handles command line arguments in a similar way, and you should add them to the end of the list you pass the function.

Receiving Output Text from Launched Commands

You can also launch terminal commands using `subprocess.run()` and `subprocess.Popen()`. You may want your Python code to receive the text output of these commands or simulate keyboard input to them. For example, let's launch the `ping` command from Python and receive the text it produces. (The details of the `ping` command are beyond the scope of this book.) On Windows, you'll use the `-n 4` arguments to send four network "ping" requests that check whether the Nostarch.com server is online. If

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

you're on macOS and Linux, replace -n with -c. This command takes a few seconds to run:

```
>>> import subprocess
>>> proc = subprocess.run(['ping', '-n', '4', 'nostarch.com'], capture_output=True, text=True)
>>> print(proc.stdout)
Pinging nostarch.com [104.20.120.46] with 32 bytes of data:
Reply from 104.20.120.46: bytes=32 time=19ms TTL=59
Reply from 104.20.120.46: bytes=32 time=17ms TTL=59
Reply from 104.20.120.46: bytes=32 time=97ms TTL=59
Reply from 104.20.120.46: bytes=32 time=217ms TTL=59

Ping statistics for 104.20.120.46:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
Approximate round trip times in milli-seconds:
    Minimum = 17ms, Maximum = 217ms, Average = 87ms
```

When you pass the capture_output=True and text=True arguments to subprocess.run(), the text output of the launched program is stored as a string in the returned CompletedProcess object's stdout ("standard output") attribute. Your Python script can use the features of other programs and then parse the text output as a string.

Running Task Scheduler, launchd, and cron

If you're computer savvy, you may know about Task Scheduler on Windows, launchd on macOS, or the cron scheduler on Linux. These well-documented and reliable tools all allow you to schedule applications to launch at specific times.

Using your operating system's built-in scheduler saves you from writing your own clock-checking code to schedule your programs. If you just need your program to pause only briefly, however, it's best to instead loop until a certain date and time, calling time.sleep(1) on each iteration through the loop.

Opening Files with Default Applications

Double-clicking a *.txt* file on your computer will automatically launch the application associated with the *.txt* file extension. Each operating system has a program that performs the equivalent of this double-clicking action. On Windows, this is the start command. On macOS and Linux, this is the open command. Enter the following into the interactive shell, passing either 'start' or 'open' to run(), depending on your system. Also, on Windows, you should pass the shell=True keyword argument, as shown here:

```
>>> file_obj = open('hello.txt', 'w')  # Create a hello.txt file.
>>> file_obj.write('Hello, world!')
13
>>> file_obj.close()
>>> import subprocess
>>> subprocess.run(['start', 'hello.txt'], shell=True)
```

Here, we write `Hello, world!` to a new *hello.txt* file. Then, we call `run()`, passing it a list containing the program or command name (in this example, `'start'` for Windows) and the filename. The operating system knows all of the file associations and can figure out that it should launch, say, *Notepad.exe* to handle the *hello.txt* file on Windows.

Project 15: Simple Countdown Program

Just as it's hard to find a simple stopwatch application, it can be hard to find a simple countdown application. Let's write a countdown program that plays an alarm at the end of the countdown.

At a high level, here's what your program will do:

- Pause for one second in between displaying each number in the countdown by calling `time.sleep()`.
- Call `subprocess.run()` to open an *alarm.wav* sound file with the default application.

Open a new file editor tab and save it as *simplecountdown.py*.

Step 1: Count Down

This program will require the `time` module for the `time.sleep()` function and the `subprocess` module for the `subprocess.run()` function. Enter the following code and save the file as *simplecountdown.py*:

```
# https://autbor.com/simplecountdown.py - A simple countdown script

import time, subprocess

❶ time_left = 60
while time_left > 0:
  ❷ print(time_left)
  ❸ time.sleep(1)
  ❹ time_left = time_left - 1

# TODO: At the end of the countdown, play a sound file.
```

After importing `time` and `subprocess`, make a variable called `time_left` to hold the number of seconds left in the countdown ❶. We set it to `60` here, but you can change the value to whatever you'd like, or even set it based on a command line argument.

In a `while` loop, display the remaining count ❷, pause for one second ❸, and then decrement the `time_left` variable ❹ before the loop starts over again. The loop will keep looping as long as `time_left` is greater than `0`. After that, the countdown will be over. Next, let's fill in the `TODO` comment with code that plays the sound file.

Step 2: Play the Sound File

While Chapter 12 covers the `playsound` module to play sound files of various formats, the quick and easy way to do this is to launch whatever application the user already uses to play sound files. The operating system will figure out from the *.wav* file extension which application it should launch to play the file. This *.wav* file could easily be some other sound file format, such as *.mp3* or *.ogg*. You can use any sound file on your computer to play at the end of the countdown, or download *alarm.wav* from *https://autbor.com/alarm.wav*.

Add the following to your code:

```
# https://autbor.com/simplecountdown.py - A simple countdown script

--snip--

# At the end of the countdown, play a sound file.
#subprocess.run(['start', 'alarm.wav'], shell=True)  # Windows
#subprocess.run(['open', 'alarm.wav'])  # macOS and Linux
```

After the `while` loop finishes, *alarm.wav* (or the sound file you choose) will play to notify the user that the countdown is over. Uncomment the `subprocess.run()` call for your operating system. On Windows, be sure to include `'start'` in the list you pass to `run()`. On macOS and Linux, pass `'open'` instead of `'start'` and remove `shell=True`.

Instead of playing a sound file, you could save a text file somewhere with a message like *Break time!* and use `subprocess.run()` to open it at the end of the countdown. This will effectively create a pop-up window with a message. Or you could use the `webbrowser.open()` function to open a specific website at the end of the countdown. Unlike some free countdown application you'd find online, your own countdown program's alarm can be anything you want!

Ideas for Similar Programs

A countdown essentially produces a simple delay before continuing the program's execution. You could use the same approach for other applications and features, such as the following:

- Use `time.sleep()` to give the user a chance to press CTRL-C to cancel an action, such as deleting files. Your program can print a "Press CTRL-C to cancel" message and then handle any `KeyboardInterrupt` exceptions with `try` and `except` statements.
- For a long-term countdown, you can use `timedelta` objects to measure the number of days, hours, minutes, and seconds until some point (a birthday? an anniversary?) in the future.

Summary

The Unix epoch (January 1, 1970, at midnight, UTC) is a standard reference time for many programming languages, including Python. While the time.time() function module returns an epoch timestamp (that is, a float value of the number of seconds since the Unix epoch), the datetime module is better for performing date arithmetic and formatting or parsing strings with date information.

The time.sleep() function will block (that is, not return) for a certain number of seconds. It can be used to add pauses to your program. But if you want to schedule your programs to start at a certain time, the instructions at *https://nostarch.com/automate-boring-stuff-python-3rd-edition* can tell you how to use the scheduler already provided by your operating system.

Finally, your Python programs can launch other applications with the subprocess.run() function. Command line arguments can be passed to the run() call to open specific documents with the application. Alternatively, you can use the start or open program with run() and use your computer's file associations to automatically figure out which application to use to open a document. By using the other applications on your computer, your Python programs can leverage their capabilities for your automation needs.

Practice Questions

1. What is the Unix epoch?
2. What function returns the number of seconds since the Unix epoch?
3. What time module function returns a human-readable string of the current time, like 'Mon Jun 15 14:00:38 2026'?
4. How can you pause your program for exactly five seconds?
5. What does the round() function return?
6. What is the difference between a datetime object and a timedelta object?
7. Using the datetime module, what day of the week was January 7, 2019?

Practice Programs

For practice, write programs to do the following tasks.

Prettified Stopwatch

Expand the stopwatch project from this chapter so that it uses the rjust() and ljust() string methods to "prettify" the output. (These methods were covered in Chapter 8.) Instead of output such as this . . .

```
Lap #1: 3.56 (3.56)
Lap #2: 8.63 (5.07)
Lap #3: 17.68 (9.05)
Lap #4: 19.11 (1.43)
```

. . . the output should look like this:

```
Lap # 1:   3.56 (  3.56)
Lap # 2:   8.63 (  5.07)
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
Lap # 3:  17.68 (  9.05)
Lap # 4:  19.11 (  1.43)
```

Next, use the pyperclip module introduced in Chapter 8 to copy the text output to the clipboard so that the user can quickly paste the output to a text file or email.

Friday the 13th Finder

Friday the 13th is considered an unlucky day by those with triskaidekaphobia (though personally I celebrate it as a lucky day). With leap years and months of varying lengths, it can be hard to figure out when the next Friday the 13th is coming.

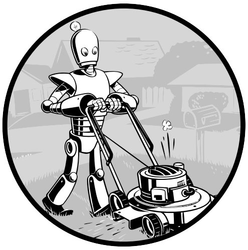
Write two programs. The first program should create a datetime object for the current day and a timedelta object of one day. If the current day is a Friday the 13th, it should print the month and year. Then, it should add the timedelta object to the datetime object to set it to the next day, and repeat the check. Have it repeat until it has found the next ten Friday the 13th dates.

The second program should do the same thing except subtract the timedelta object. This program will find all of the past months and years with a Friday the 13th, and stop when it reaches the year 1.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

20

SENDING EMAIL, TEXTS, AND PUSH NOTIFICATIONS



Checking and replying to email is a huge time sink, and you can't just write a program to handle all your email for you, as each message requires its own response. But you can still automate plenty of email-related tasks once you know how to write programs that can send and receive email.

For example, maybe you have a spreadsheet full of customer records and want to send each customer a different form letter depending on their age and location details. Commercial software might not be able to do this for you. Fortunately, you can write your own program to send these emails, saving yourself a lot of time spent copying and pasting.

You can also write programs to send SMS text messages and push notifications to notify you of things even while you're away from your computer. If you're automating a task that takes a couple of hours to do, you probably don't want to go back to your computer every few minutes to check on the program's status. Instead, the program can just text your phone when it's done, freeing you to focus on more important things while you're away from your computer.

This chapter features the EZGmail module, a simple way to send and read emails from Gmail accounts, as well as the free *ntfy* service that provides push notifications between devices.

WARNING *I highly recommend that you set up a separate email account for any scripts that send or receive emails. This will prevent bugs in your programs from affecting your personal email account (by deleting emails or accidentally spamming your contacts, for example). It's a good idea to first do a dry run by commenting out the code that actually sends or deletes emails and replacing it with a temporary* `print()` *call. This way, you can test your program before running it for real.*

The Gmail API

Gmail owns close to one-third of the email client market share, and most likely you have at least one Gmail email address. Because of Gmail's additional security and anti-spam measures, controlling a Gmail account is better done through the EZGmail module than through the `smtplib` and `imaplib` modules in Python's standard library. I wrote EZGmail to work on top of the official Gmail API and provide functions that make it easy to use Gmail from Python. For full instructions on installing EZGmail, see Appendix A.

Enabling the API

Before you write code, you must first sign up for a Gmail email account at *https://gmail.com*. Then, you must set up the Gmail API for your account through the Google Cloud console at *https://console.cloud.google.com*. These steps are identical to the steps for setting up EZSheets detailed in Chapter 15, so I won't repeat them in this chapter. Create a new project and make sure to enable the Gmail API instead of the Google Sheets API. On step 2 of the OAuth consent screen configuration, add the *https://mail.google.com* scope to let your Python scripts read and send email. When you're done, you should have a credentials file and a token file.

Then, in the interactive shell, enter the following code:

```
>>> import ezgmail
>>> ezgmail.init()
```

If no error appears, EZGmail has been correctly installed.

Sending Mail

Once EZGmail is configured, you should be able to send email with a single function call:

```
>>> import ezgmail
>>> ezgmail.send('recipient@example.com', 'Subject line', 'Body of the email')
```

If you want to attach files to your email, you can provide an extra list argument to the `send()` function:

```
>>> ezgmail.send('recipient@example.com', 'Subject line', 'Body of the email',
['attachment1.jpg', 'attachment2.mp3'])
```

Note that as part of its security and anti-spam features, Gmail might not send repeated emails with the exact same text (as these are likely spam) or emails that contain *.exe* or *.zip* file attachments (as they are potentially viruses).

You can also supply the optional keyword arguments `cc` and `bcc` to send carbon copies and blind carbon copies:

```
>>> import ezgmail
>>> ezgmail.send('recipient@example.com', 'Subject line', 'Body of the
email', cc='friend@example.com', bcc='otherfriend@example.com,
someoneelse@example.com')
```

If you need to remember which Gmail address the *token.json* file is configured for, you can examine `ezgmail.EMAIL_ADDRESS`:

```
>>> import ezgmail
>>> ezgmail.EMAIL_ADDRESS
'example@gmail.com'
```

Be sure to treat the *token.json* file in the same way as your password. If someone else obtains this file, they can access your Gmail account (though they won't be able to change your Gmail password). To revoke previously issued *token.json* files, return to the Google Cloud console and delete the credential for the compromised token. You will need to repeat the setup steps to generate a new credential and token file before you can resume using EZGmail.

Reading Mail

Gmail organizes emails that are replies to each other into conversation threads. When you log in to Gmail in your web browser or through an app, you're really looking at email threads rather than individual emails (even if the thread has only one email in it).

EZGmail has `GmailThread` and `GmailMessage` objects to represent conversation threads and individual emails, respectively. A `GmailThread` object has a `messages` attribute that holds a list of `GmailMessage` objects. The `unread()` function returns a list of `GmailThread` objects for the 25 most recent unread emails, which can then be passed to `ezgmail.summary()` to print a summary of the conversation threads in that list:

```
>>> import ezgmail
>>> unread_threads = ezgmail.unread()  # List of GmailThread objects
>>> ezgmail.summary(unread_threads)
Al, Jon - Do you want to watch RoboCop this weekend? - Dec 09
Jon - Thanks for stopping me from buying Bitcoin. - Dec 09
```

The summary() function is handy for displaying a quick summary of the email threads, but to access specific messages (and parts of messages), you'll want to examine the messages attribute of the GmailThread object. The messages attribute contains a list of the GmailMessage objects that make up the thread, and these have subject, body, timestamp, sender, and recipient attributes that describe the email:

```
>>> len(unread_threads)
2
>>> str(unread_threads[0])
"<GmailThread len=2 snippet= Do you want to watch RoboCop this weekend?'>"
>>> len(unread_threads[0].messages)
2
>>> str(unread_threads[0].messages[0])
"<GmailMessage from='Al Sweigart <al@inventwithpython.com>' to='Jon Doe
<example@gmail.com>' timestamp=datetime.datetime(2026, 12, 9, 13, 28, 48)
subject='RoboCop' snippet='Do you want to watch RoboCop this weekend?'>"
>>> unread_threads[0].messages[0].subject
'RoboCop'
>>> unread_threads[0].messages[0].body
'Do you want to watch RoboCop this weekend?\r\n'
>>> unread_threads[0].messages[0].timestamp
datetime.datetime(2026, 12, 9, 13, 28, 48)
>>> unread_threads[0].messages[0].sender
'Al Sweigart <al@inventwithpython.com>'
>>> unread_threads[0].messages[0].recipient
'Jon Doe <example@gmail.com>'
```

To retrieve more than the 25 most recent unread emails, pass an integer for the maxResults keyword argument. For example, ezgmail.unread(maxResults=50) will return the 50 most recent unread emails.

Like the ezgmail.unread() function, the ezgmail.recent() function will return the 25 most recent threads in your Gmail account:

```
>>> recent_threads = ezgmail.recent()
>>> len(recent_threads)
25
>>> recent_threads = ezgmail.recent(maxResults=100)
>>> len(recent_threads)
46
```

You can pass an optional maxResults keyword argument to change this limit.

Searching for Mail

In addition to using ezgmail.unread() and ezgmail.recent(), you can search for specific emails, the same way you would if you entered queries into the Gmail search box, by calling ezgmail.search():

```
>>> result_threads = ezgmail.search('RoboCop')
>>> len(result_threads)
```

```
1
>>> ezgmail.summary(result_threads)
Al, Jon - Do you want to watch RoboCop this weekend? - Dec 09
```

The previous `search()` call should yield the same results as if you had entered *RoboCop* into the search box, as in Figure 20-1.

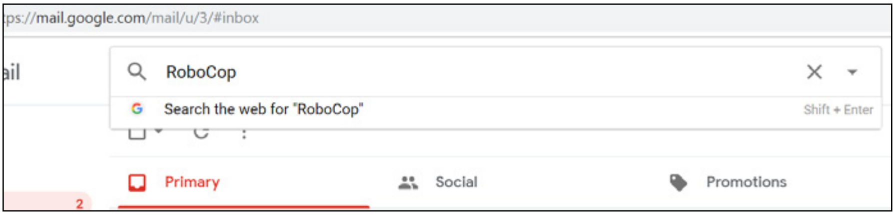

ps://mail.google.com/mail/u/3/#inbox
ail
RoboCop
Search the web for "RoboCop"
Shift + Enter
Primary
Social
Promotions
2

Figure 20-1: Searching for RoboCop *emails at the Gmail website*

Like `unread()` and `recent()`, the `search()` function returns a list of `GmailThread` objects. You can also pass to the `search()` function any of the special search operators that you can enter into the search box, such as the following:

'label:UNREAD' For unread emails

'from:al@inventwithpython.com' For emails from *al@inventwithpython.com*

'subject:hello' For emails with "hello" in the subject

'has:attachment' For emails with file attachments

You can view a full list of search operators at *https://support.google.com/mail/answer/7190.*

Downloading Attachments

A `GmailMessage` object has an `attachments` attribute that is a list of filenames for the message's attached files. You can pass any of these names to a `GmailMessage` object's `downloadAttachment()` method to download the files. You can also download all of them at once with `downloadAllAttachments()`. By default, EZGmail saves attachments to the current working directory, but you can pass an additional `downloadFolder` keyword argument to `downloadAttachment()` and `downloadAllAttachments()` as well. Here is an example:

```
>>> import ezgmail
>>> threads = ezgmail.search('vacation photos')
>>> threads[0].messages[0].attachments
['tulips.jpg', 'canal.jpg', 'bicycles.jpg']
>>> threads[0].messages[0].downloadAttachment('tulips.jpg')
>>> threads[0].messages[0].downloadAllAttachments(downloadFolder='vacation2026')
['tulips.jpg', 'canal.jpg', 'bicycles.jpg']
```

If a file already exists with the attachment's filename, the downloaded attachment will automatically overwrite it.

EZGmail contains additional features, and you can find the full documentation at *https://github.com/asweigart/ezgmail.*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

SMS Email Gateways

People are more likely to be near their smartphones than their computers, so text messages are often a more reliable way of sending immediate notifications than email. Also, text messages are usually shorter, making it more likely that a person will get around to reading them. The easiest, though not most reliable, way to send text messages is by using a *short message service (SMS) email gateway*, an email server that a cell phone provider has set up to receive texts via email and then forward them to the recipient as text messages.

You can write a program to send these emails using EZGmail or the `smtplib` module. The phone number and phone company's email server make up the recipient email address. For example, to send a text to a Verizon customer with the phone number 212-555-1234, you would send an email to *2125551234@vtext.com*. The subject and body of the email would appear in the body of the text message.

You can find the SMS email gateway for a cell phone provider by doing a web search for "sms email gateway *provider name*." Table 20-1 lists the gateways for several popular providers. Many providers have separate email servers for SMS, which limits messages to 160 characters, and Multimedia Messaging Service (MMS), which has no character limit. If you wanted to send a photo, you would have to use the MMS gateway and attach the file to the email.

If you don't know the recipient's cell phone provider, you can try searching for it using a carrier lookup site. The best way to find these sites is by searching the web for "find cell phone provider for number." Many of these sites will let you look up numbers for free (though they will charge you to look up hundreds or thousands of phone numbers through their API).

Table 20-1: SMS Email Gateways for Cell Phone Providers

Cell phone provider	SMS gateway	MMS gateway
AT&T	*number@txt.att.net*	*number@mms.att.net*
Boost Mobile	*number@sms.myboostmobile.com*	Same as SMS
Cricket	*number@sms.cricketwireless.net*	*number@mms.cricketwireless.net*
Google Fi	*number@msg.fi.google.com*	Same as SMS
Metro PCS	*number@mymetropcs.com*	Same as SMS
Republic Wireless	*number@text.republicwireless.com*	Same as SMS
Sprint (now T-Mobile)	*number@messaging.sprintpcs.com*	*number@pm.sprint.com*
T-Mobile	*number@tmomail.net*	Same as SMS
U.S. Cellular	*number@email.uscc.net*	*number@mms.uscc.net*
Verizon	*number@vtext.com*	*number@vzwpix.com*
Virgin Mobile	*number@vmobl.com*	*number@vmpix.com*
Xfinity Mobile	*number@vtext.com*	*number@mypixmessages.com*

While SMS email gateways are free and simple to use, there are several major disadvantages to them:

- You have no guarantee that the text will arrive promptly, or at all.
- You have no way of knowing if the text failed to arrive.
- The text recipient has no way of replying.
- SMS gateways may block you if you send too many emails, and there's no way to find out how many is "too many."
- The fact that the SMS gateway delivers a text message today doesn't mean it will work tomorrow.

Sending texts via an SMS gateway is ideal when you need to transmit the occasional nonurgent message. If you want a more reliable way to send SMS text messages, especially in bulk, you can use the API of a telecom service provider such as Twilio. These services often require a subscription or usage fees, and you may need to submit an application to use them. Regulations may differ for each country and change over time.

An alternative to sending SMS texts is to use a free push notification service, as the next section will explain.

Push Notifications

HTTP *pub-sub notification services* allow you to send and receive short, disposable messages over the internet via HTTP web requests. Chapter 13 covers the use of the Requests library to make HTTP requests, and we'll use it here to interact with the free online service ntfy (pronounced *notify* and always written in lowercase) at *https://ntfy.sh*. The ntfy service is free and doesn't require any sign-up or registration.

Before we get started, install the ntfy app on your mobile phone so that you can receive notifications. This app is free and can be found in the app stores for Android and iPhone. You can also receive notifications in your web browser by going to *https://ntfy.sh/app*.

These apps check with the ntfy service for any messages sent to a topic. You can think of a topic as a chat room or group chat name. Anyone in the world can send messages to a topic, and anyone in the world can subscribe to a topic to read these messages. If you want to send messages to just yourself, use a secret topic with random letters. Treat this topic name as a password and share it only with those you intend to read the messages. This chapter will use the topic `AlSweigartZPgxBQ42` in the example code, though I recommend you use your own secret topic that contains a suffix of random letters and numbers. Topics are case-sensitive, and even if you keep the topic a secret, do not use ntfy to send sensitive information such as passwords or credit card numbers.

Sending Notifications

Sending a push notification to everyone who is subscribed to a topic requires nothing more than making an HTTP request to the ntfy web server. This means it can be done entirely with the Requests library. You don't need to install a ntfy-specific package.

To send the request, enter the following into the interactive shell. Replace the `AlSweigartZPgxBQ42` example topic used throughout this chapter with your own random, secret topic:

```
>>> import requests
>>> requests.post('https://ntfy.sh/AlSweigartZPgxBQ42', 'Hello, world!')
<Response [200]>
```

Note that we call `requests.post()` to make a POST HTTP request to send a notification. This is different from the `requests.get()` function covered in Chapter 13 to download web pages.

Anyone subscribed to the topic `AlSweigartZPgxBQ42` will receive the message "Hello, world!" within a few seconds (though sometimes messages may be delayed by a few minutes). You can also view these yourself at *https://ntfy.sh/AlSweigartZPgxBQ42*.

The ntfy service has some limitations. Free users are limited to 250 messages per day, and messages can be 4,096 bytes in size at most. Flooding messages to various topics may result in your IP address becoming temporarily blocked. You can obtain a paid account to increase these limits on the ntfy website. Paid ntfy accounts can set reserved topics and limit who posts to them. If you don't have permission to post to a reserved topic, you'll get a `<Response [403]>` response to your `requests.post()` function call.

Within the 4,096-byte limit, your messages can take on any format. Note that there is no way to determine who posted a message to a topic, so you may want to include "To" and "From" labels within the text of your message. Better yet, you could do so using JSON or some other data serialization format covered in Chapter 18.

If you want your Python programs to send you a notification, these are the only two lines of code you need once you have the ntfy app installed on your phone and have subscribed to the topic. You're free to run your Python program and step away for coffee, knowing that you'll receive a notification on your phone when your program has finished its boring task.

Transmitting Metadata

While the message you send is a freeform text string value, ntfy can optionally attach metadata values, such as a title, a priority level, and tags, to each message.

A *title* is similar to an email subject line, and most apps display it above the message text in a larger font. A *priority level* ranges from 1 (the lowest priority) to 5 (the highest), with 3 being the default. A higher priority doesn't deliver messages any faster; it just allows subscribers to configure their notification apps to display only messages of a certain priority or higher. *Tags* are keywords that subscribers can use to filter messages. Tags can also be the name of an emoji to display next to the message title. You can find a list of these emojis at *https://docs.ntfy.sh/publish/#tags-emojis*.

This metadata is included in the HTTP request as headers, so you'll need to pass a dictionary of them to the `headers` keyword argument. Enter the following into the interactive shell to post a message with metadata:

```
>>> import requests
>>> requests.post('https://ntfy.sh/AlSweigartZPgxBQ42', 'The rent is too high!',
headers={'Title':'Important: Read this!', 'Tags': 'warning,neutral_face', 'Priority':'5'})
<Response [200]>
```

These features are useful for human users reading the notifications on their phone app. However, you can also write code so that Python scripts can receive push notifications, as we'll discuss in the next section.

Receiving Notifications

Your Python programs can also read the messages posted to a particular topic by making HTTP requests with the Requests library. Send a notification message using the code in the previous sections, and then enter the following into the interactive shell using the same topic as the notifications:

```
>>> import requests
>>> resp = requests.get('https://ntfy.sh/AlSweigartZPgxBQ42/json?poll=1')
>>> resp.text
'{"id":"1jnHKeFNqwnS","time":1797823340,"expires":1797866540,"event":
"message","topic":"AlSweigartZPgxBQ42","message":"Hello, world!"}\n
{"id":"wZ22cjyKXw1F","time":1797823712,"expires":1797866912,"event":
"message","topic":"AlSweigartZPgxBQ42","title":"Important: Read this!",
"message":"The rent is too high!","priority":5,"tags":["warning",
"neutral_face"]}\n'
```

Note that we call the `requests.get()` function to receive notifications, unlike the `requests.post()` function used when sending notifications. Also, the URL ends in `/json?poll=1`.

This is retrieving messages through polling, which returns all the messages for a topic that the server has. There are also streaming methods for retrieving ntfy messages, but polling has the simplest code. You can also add a *since* URL parameter after *poll=1* to get the messages by one of the following criteria:

since=10m Retrieves all messages for the topic in the last 10 minutes. You can also use *s* for seconds and *h* for hours, such as *since=2h30m* for all messages in the last two and a half hours.

since=1737866912 Retrieves all messages since the Unix epoch timestamp of `1737866912`. This kind of timestamp is returned by `time.time()` and represents the number of seconds since January 1, 1970. Chapter 19 covers time-related functions.

since=wZ22cjyKXw1F Retrieves all messages after the message that had the ID of `'wZ22cjyKXw1F'`.

Separate additional URL parameters by an ampersand (&). For example, passing the URL *https://ntfy.sh/AlSweigartZPgxBQ42/json?poll=1&since=10m* retrieves all messages for the *AlSweigartZPgxBQ42* topic in the last 10 minutes. To reduce the load on the ntfy server, you should poll only once a minute or once every few minutes rather than as fast as possible in an infinite loop. If you need to receive notifications immediately, consult the online documentation to read about subscribing to notification streams.

The text of this HTTP response is not valid JSON, since it contains a JSON object on each line of text, rather than one JSON object, so we use the `splitlines()` string method before parsing them individually with the `json` module (as covered in Chapter 18). Continue the previous interactive shell example:

```
>>> import json
>>> notifications = []
>>> for json_text in resp.text.splitlines():
...     notifications.append(json.loads(json_text))
...
>>> notifications[0]['message']
'Hello, world!'
>>> notifications[1]['message']
'The rent is too high!'
```

The `json.loads()` function converts the JSON text from ntfy into a Python dictionary. Let's look at each of the key value pairs:

`"id":"wZ22cjyKXw1F"` The `'id'` key's value is a unique identification string that can help differentiate multiple notifications even if they have the same text.

`"time":1797823712` The `'time'` key's value is a Unix epoch timestamp of when the notification was created. Calling `str(datetime.datetime.fromtimestamp(1797823712))` returns the human-readable string `'2026-12-20 21:28:32'`.

`"expires":1797866912` The `'expires'` key's value is a Unix epoch timestamp of when the notification will be deleted from the ntfy server.

`"event":"message"` The `'event'` key's value can be either `'message'`, `'open'`, `'keepalive'`, or `'poll_request'`. These event types are explained in the online documentation, but for now, you're probably only interested in `'message'` events.

`"topic":"AlSweigartZPgxBQ42"` The topic part of the URL is repeated in the `'topic'` key's value.

`"title":"Important: Read this!"` If the notification has a title, there will be a `'title'` key with it as a string value.

`"message":"The rent is too high!"` The `'message'` key's value is a string of the notification's text.

`"priority":5` If the notification has a priority, there will be a `'priority'` key with an integer value from 1 to 5.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

"tags":["warning","neutral_face"] @RunInPara:If the notification has tags, there will be a `'tags'` key with it as a list of string values. These string values may be the names of emoji characters to display.

By reading the values in this dictionary, your Python programs can use the Requests library to receive notifications made by users or other Python scripts. The ntfy service is one of the easiest ways to make programs that can communicate with each other over the internet (though keep in mind the limit of 250 messages per day for free users).

Summary

We communicate with each other over the internet and cell phone networks in dozens of different ways, but email and texting predominate. Your programs can communicate through these channels, which gives them powerful new notification features.

As a security and spam precaution, some popular email services like Gmail don't allow you to use the standard SMTP and IMAP protocols to access their services. The EZGmail package acts as a convenient wrapper for the Gmail API, letting your Python scripts access your Gmail account. I highly recommend that you set up a separate Gmail account for your scripts to use so that potential bugs in your program don't cause problems for your personal Gmail account.

Texting is a bit different from email, since, unlike with email, you'll need more than just an internet connection to send SMS texts. You can use SMS email gateways to send texts from an email account, though this requires you to know the phone user's telecom carrier and is not a reliable way to send messages. If you're only sending short messages to yourself, you can use the push notification system at *https://ntfy.sh*, then install the ntfy app on your phone to have your Python scripts send messages to topic subscribers.

With these modules in your skill set, you'll be able to program the specific conditions under which your programs should send notifications or reminders. Now your programs will have a reach that's far beyond the computer they're running on!

Practice Questions

1. When using the Gmail API, what are the credentials and token files?
2. In the Gmail API, what's the difference between "thread" and "message" objects?
3. Using `ezgmail.search()`, how can you find emails that have file attachments?
4. What are some of the disadvantages of using an SMS email gateway to send text messages?
5. What Python library can send and receive notifications to ntfy?

Practice Programs

For practice, write programs to do the following tasks.

Umbrella Reminder

Chapter 13 showed you how to use the `requests` module to scrape data from *https://weather.gov*. Write a program that runs just before you wake up in the morning and checks whether rain is in the forecast for that day. If so, have the program text you a reminder to pack an umbrella before leaving the house.

Auto Unsubscriber

Write a program that scans your email account, finds all the unsubscribe links in all your emails, and automatically opens them in a browser. This program will have to log in to your Gmail account. You can use Beautiful Soup (covered in Chapter 13) to check for any instance where the word *unsubscribe* occurs within an HTML link tag. Once you have a list of these URLs, you can use `webbrowser.open()` to automatically open all of these links in a browser.

You'll still have to manually go through and complete any additional steps to unsubscribe yourself from these lists. In most cases, this involves clicking a link to confirm. But this script saves you from having to go through all of your emails looking for unsubscribe links.

Email-Based Computer Control

Write a program that checks an email or ntfy account every 15 minutes for any instructions you send it and executes those instructions automatically. For example, BitTorrent is a peer-to-peer downloading system. Using free BitTorrent software such as qBittorrent, you can download large media files on your home computer. If you send the program a (completely legal, not at all piratical) BitTorrent link, the program will eventually check its email or look for ntfy notifications, find this message, extract the link, and then launch qBittorrent to start downloading the file. This way, you can have your home computer begin downloads while you're away, and finish the (completely legal, not at all piratical) download by the time you return home.

Chapter 19 covered how to launch programs on your computer using the `subprocess.Popen()` function. For example, the following call would launch the qBittorrent program, along with a torrent file:

```
qbProcess = subprocess.Popen(['C:\\Program Files (x86)\\qBittorrent\\
qbittorrent.exe', 'shakespeare_complete_works.torrent'])
```

Of course, you'll want the program to make sure the emails come from you. In particular, you might want to require that the emails contain a password, since it is fairly trivial for hackers to fake a "from" address in emails. The program should delete the emails it finds so that it doesn't repeat instructions every time it checks the email account. As an extra feature,

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

have the program email or text you a confirmation every time it executes a command. Since you won't be sitting in front of the computer that is running the program, it's a good idea to use the logging functions (see Chapter 5) to write a text file log that you can check if errors come up.

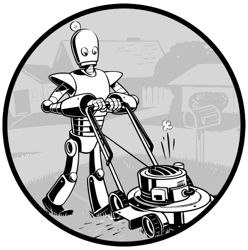
The qBittorrent program (as well as other BitTorrent applications) has a feature that enables it to quit automatically after the download completes. Chapter 19 explained how you can determine when a launched application has quit with the `wait()` method for `Popen` objects. The `wait()` method call will block until qBittorrent has stopped, and then your program can email or text you a notification that the download has completed.

There are plenty of possible features you could add to this project. If you get stuck, you can download an example implementation of this program from *https://nostarch.com/automate-boring-stuff-python-3rd-edition*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

21

MAKING GRAPHS AND MANIPULATING IMAGES



If you own a digital camera or upload photos from your phone to a social media site, you probably cross paths with digital image files all the time. You may know how to use basic graphics software such as Microsoft Paint or Paintbrush, or even more advanced applications such as Adobe Photoshop. But if you need to edit a massive number of images, altering them by hand can be a lengthy, boring job.

Enter Pillow, a third-party Python package for interacting with image files. This package has several functions that make cropping, resizing, and editing the content of an image easy. This chapter covers the use of Pillow to enable Python to automatically edit hundreds or thousands of images with ease.

This chapter also covers Matplotlib, a popular library for making professional-looking graphs. Matplotlib is rich in features and customizable options, and there are many books entirely dedicated to it. Here, we'll cover the basics of generating graph images with Matplotlib.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Computer Image Fundamentals

To manipulate an image, you must understand how to work with colors and coordinates in Pillow. You can install the latest version of Pillow by following the instructions in Appendix A.

Colors and RGBA Values

Computer programs often represent a color in an image as an *RGBA value*, a group of numbers that specify the amount of red, green, blue, and *alpha* (or transparency) to include. Each of these component values is an integer ranging from 0 (none at all) to 255 (the maximum). These RGBA values belong to individual *pixels*, the smallest dot of a single color the computer screen can show. A pixel's RGB setting tells it precisely what shade of color it should display. If an image on the screen is superimposed over a background image or desktop wallpaper, the alpha value determines how much of the background you can "see through" the image's pixel.

Pillow represents RGBA values using a tuple of four integer values. For example, it represents the color red with `(255, 0, 0, 255)`. This color has the maximum amount of red, no green or blue, and the maximum alpha value, meaning it's fully opaque. Pillow represents green with `(0, 255, 0, 255)` and blue with `(0, 0, 255, 255)`. White, the combination of all colors, is `(255, 255, 255, 255)`, while black, which has no color at all, is `(0, 0, 0, 255)`.

If a color has an alpha value of `0`, it is invisible, and it doesn't really matter what the RGB values are. After all, invisible red looks the same as invisible black.

Pillow uses the same standard color names as HTML. Table 21-1 lists a selection of standard color names and their values.

Table 21-1: Standard Color Names and Their RGBA Values

Name	RGBA value	Name	RGBA value
White	`(255, 255, 255, 255)`	Red	`(255, 0, 0, 255)`
Green	`(0, 255, 0, 255)`	Blue	`(0, 0, 255, 255)`
Gray	`(128, 128, 128, 255)`	Yellow	`(255, 255, 0, 255)`
Black	`(0, 0, 0, 255)`	Purple	`(128, 0, 128, 255)`

Pillow offers the `ImageColor.getcolor()` function so that you don't have to memorize RGBA values for the colors you want to use. This function takes a color name string as its first argument and the string `'RGBA'` as its second argument, and it returns an RGBA tuple. To see how this function works, enter the following into the interactive shell:

```
❶ >>> from PIL import ImageColor
❷ >>> ImageColor.getcolor('red', 'RGBA')
(255, 0, 0, 255)
❸ >>> ImageColor.getcolor('RED', 'RGBA')
(255, 0, 0, 255)
```

```
>>> ImageColor.getcolor('Black', 'RGBA')
(0, 0, 0, 255)
>>> ImageColor.getcolor('chocolate', 'RGBA')
(210, 105, 30, 255)
>>> ImageColor.getcolor('CornflowerBlue', 'RGBA')
(100, 149, 237, 255)
```

First, import the `ImageColor` module from `PIL` (not from Pillow, due to naming history beyond the scope of this book) ❶. The color name string you pass to `ImageColor.getcolor()` is case-insensitive, so `'red'` ❷ and `'RED'` ❸ give you the same RGBA tuple. You can also pass more unusual color names, like `'chocolate'` and `'CornflowerBlue'`.

Pillow supports a huge number of color names, from `'aliceblue'` to `'yellowgreen'`. Enter the following into the interactive shell to view the color names:

```
>>> from PIL import ImageColor
>>> list(ImageColor.colormap)
['aliceblue', 'antiquewhite', 'aqua', ... 'yellow', 'yellowgreen']
```

You can find the full list of more than 100 standard color names in the keys of the `ImageColor.colormap` dictionary.

Coordinates and Box Tuples

Image pixels are addressed with x- and y-coordinates, which respectively specify a pixel's horizontal and vertical locations in an image. The *origin* is the pixel at the top-left corner of the image and is specified with the notation (0, 0). The first zero represents the x-coordinate, which starts at zero at the origin and increases from left to right. The second zero represents the y-coordinate, which starts at zero at the origin and increases down the image. This bears repeating: y-coordinates increase going downward, which is the opposite of how you may remember y-coordinates being used in math class. Figure 21-1 demonstrates how this coordinate system works.

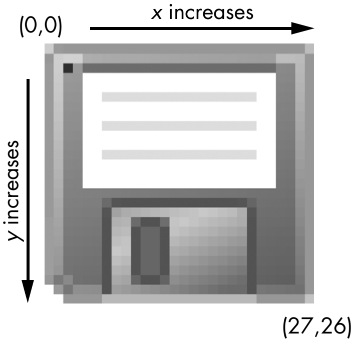


Figure 21-1: The x- and y-coordinates of a 28 × 27 image of some sort of ancient data storage device

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Many of Pillow's functions and methods take a *box tuple* argument. This means Pillow is expecting a tuple of four integer coordinates that represent a rectangular region in an image. The four integers are, in order, as follows:

Left The x-coordinate of the leftmost edge of the box.

Top The y-coordinate of the top edge of the box.

Right The x-coordinate of one pixel to the right of the rightmost edge of the box. This integer must be greater than the left integer.

Bottom The y-coordinate of one pixel lower than the bottom edge of the box. This integer must be greater than the top integer.

Note that the box includes the left and top coordinates and goes up to but does not include the right and bottom coordinates. For example, the box tuple `(3, 1, 9, 6)` represents all the pixels in the black box in Figure 21-2.

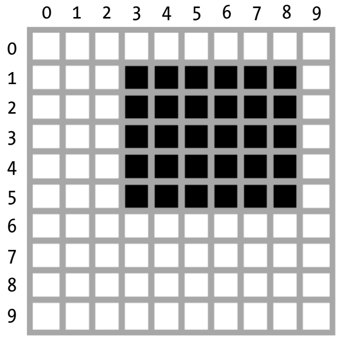


Figure 21-2: The area represented by the box tuple `(3, 1, 9, 6)`

Now that you know how colors and coordinates work in Pillow, let's use Pillow to manipulate an image.

Manipulating Images with Pillow

To practice working with Pillow, we'll use the *zophie.png* image file shown in Figure 21-3. You can download it from the book's online resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*. Save the file in your current working directory, and then load the image into Python, like so:

```
>>> from PIL import Image
>>> cat_im = Image.open('zophie.png')
>>> cat_im.show()
```

Import the `Image` module from Pillow and call `Image.open()`, passing it the image's filename. You can then store the loaded image in a variable like `cat_im`. Pillow `Image` objects have a `show()` method that opens the image in a window. This is useful when you're debugging your programs and need to identify the image in an Image object.

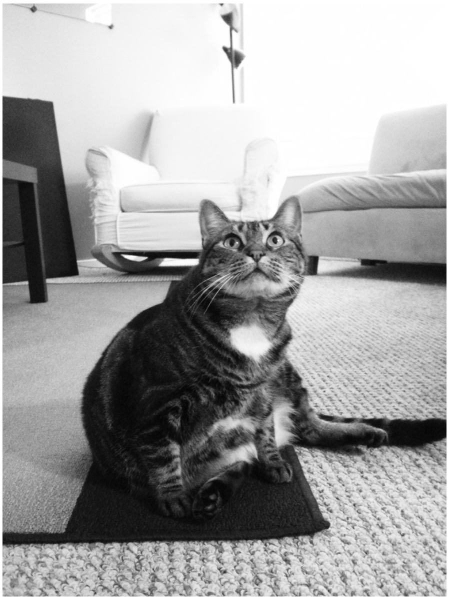


Figure 21-3: My cat, Zophie

If the image file isn't in the current working directory, change the working directory to the folder that contains the image file by calling the os.chdir() function:

```
>>> import os
>>> os.chdir('C:\\folder_with_image_file')
```

The Image.open() function returns a value of the Image object data type, which Pillow uses to represent an image as a Python value. You can load an Image object from an image file of any format by passing the Image.open() function a string of the filename. You can save any changes you make to the Image object to an image file (also of any format) with the save() method. All the rotations, resizing, cropping, drawing, and other image manipulations will occur through method calls on this Image object.

To shorten the examples in this chapter, I'll assume you've imported Pillow's Image module and stored the Zophie image in a variable named cat_im. Be sure that the *zophie.png* file is in the current working directory so that the Image.open() function can find it. Otherwise, you'll have to specify the full absolute path in the function's string argument.

Working with the Image Data Type

An Image object has several useful attributes that give you basic information about the image file from which it was loaded: its width and height, the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

filename, and the graphics format (such as JPEG, WebP, GIF, or PNG). For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> cat_im = Image.open('zophie.png')
>>> cat_im.size
❶ (816, 1088)
❷ >>> width, height = cat_im.size
❸ >>> width
816
❹ >>> height
1088
>>> cat_im.filename
'zophie.png'
>>> cat_im.format
'PNG'
>>> cat_im.format_description
'Portable network graphics'
❺ >>> cat_im.save('zophie.jpg')
```

After you've created an Image object from *zophie.png* and stored the Image object in cat_im, the object's size attribute contains a tuple of the image's width and height in pixels ❶. You can assign the values in the tuple to width and height variables ❷ in order to access the width ❸ and height ❹ individually. The filename attribute describes the original file's name. The format and format_description attributes are strings that describe the image format of the original file (with format_description being a bit more verbose).

Finally, calling the save() method and passing it 'zophie.jpg' saves a new image with the filename *zophie.jpg* to your hard drive ❺. Pillow sees that the file extension is *.jpg* and automatically saves the image using the JPEG image format. Now you should have two images, *zophie.png* and *zophie.jpg*, on your hard drive. While these files are based on the same image, they are not identical, because of their different formats.

Pillow also provides the Image.new() function, which returns an Image object—much like Image.open(), except the image represented by Image.new()'s object will be blank. The arguments to Image.new() are as follows:

- The string 'RGBA', which sets the color mode to RGBA. (There are other modes that this book doesn't go into.)
- The size as a two-integer tuple of the new image's width and height.
- The background color that the image should start with, as a four-integer tuple of an RGBA value. You can use the return value of the ImageColor.getcolor() function for this argument. Alternatively, you can pass the standard color name as a string.

For example, enter the following into the interactive shell:

```
>>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 200), 'purple')
>>> im.save('purpleImage.png')
```

```
❷ >>> im2 = Image.new('RGBA', (20, 20))
  >>> im2.save('transparentImage.png')
```

Here, we create an `Image` object for an image that's 100 pixels wide and 200 pixels tall, with a purple background ❶. We then save this image to the file *purpleImage.png*. We call `Image.new()` again to create another `Image` object, this time passing `(20, 20)` for the dimensions and nothing for the background color ❷. Invisible black, `(0, 0, 0, 0)`, is the default color used if no color argument is specified, so the second image has a transparent background. We save this 20 × 20 transparent square in *transparentImage.png*.

Cropping Images

Cropping an image means selecting a rectangular region inside an image and removing everything outside the rectangle. The `crop()` method of `Image` objects takes a box tuple and returns an `Image` object representing the cropped image. The cropping doesn't happen in place—that is, the original `Image` object is left untouched, and the `crop()` method returns a new one. Remember that a box tuple (in this case, the cropped section) includes the left column and top row of pixels but not the right column and bottom row of pixels.

Enter the following into the interactive shell:

```
>>> from PIL import Image
>>> cat_im = Image.open('zophie.png')
>>> cropped_im = cat_im.crop((335, 345, 565, 560))
>>> cropped_im.save('cropped.png')
```

This code makes a new `Image` object for the cropped image, stores the object in `cropped_im`, and then calls `save()` on `cropped_im` to save the cropped image in *cropped.png*, shown in Figure 21-4.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

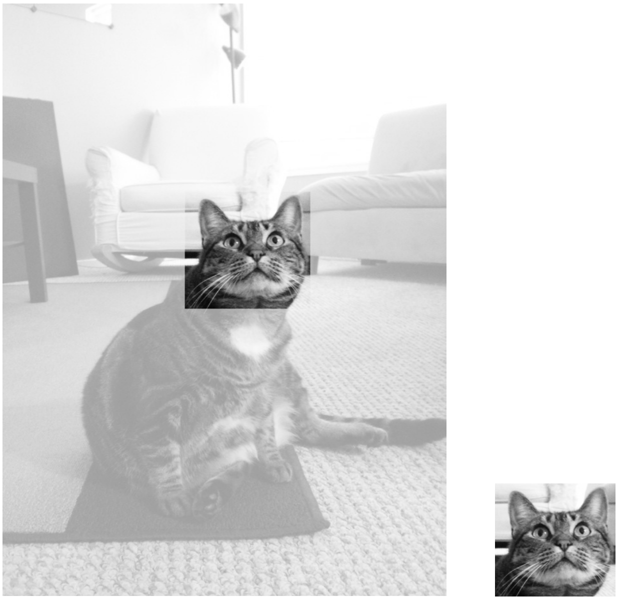


Figure 21-4: The new image is the cropped section of the original image.

Cropping creates the new file from the original.

Pasting Images onto Other Images

The `copy()` method will return a new `Image` object containing the same image as the `Image` object on which it was called. This is useful if you need to make changes to an image but also want to keep an untouched version of the original. For example, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> cat_im = Image.open('zophie.png')
>>> cat_copy_im = cat_im.copy()
```

The `cat_im` and `cat_copy_im` variables contain two separate `Image` objects, which both have the same image on them. Now that you have an `Image` object stored in `cat_copy_im`, you can modify `cat_copy_im` as you like and save it to a new filename, leaving `cat_im` untouched.

When called on an `Image` object, the `paste()` method pastes another image on top of it. Let's continue the shell example by pasting a smaller image onto `cat_copy_im`:

```
>>> face_im = cat_im.crop((335, 345, 565, 560))
>>> face_im.size
(230, 215)
>>> cat_copy_im.paste(face_im, (0, 0))
>>> cat_copy_im.paste(face_im, (400, 500))
>>> cat_copy_im.save('pasted.png')
```

First, we pass `crop()` a box tuple for the rectangular area in *zophie.png* that contains Zophie's face. This method call creates an `Image` object representing a 230 × 215 crop, which we store in `face_im`. Now we can paste `face_im` onto `cat_copy_im`. The `paste()` method takes two arguments: a source `Image` object and a tuple of the x- and y-coordinates where we want to paste the top-left corner of the source `Image` object onto the main `Image` object. Here, we call `paste()` twice on `cat_copy_im`, pasting two copies of `face_im` onto `cat_copy_im`. Finally, we save the modified `cat_copy_im` to *pasted.png*, shown in Figure 21-5.

NOTE *Despite their names, the `copy()` and `paste()` methods in Pillow don't use your computer's clipboard.*

The `paste()` method modifies its `Image` object *in place*; it doesn't return an `Image` object with the pasted image. If you want to call `paste()` but also keep an untouched version of the original image around, you'll need to first copy the image and then call `paste()` on that copy.

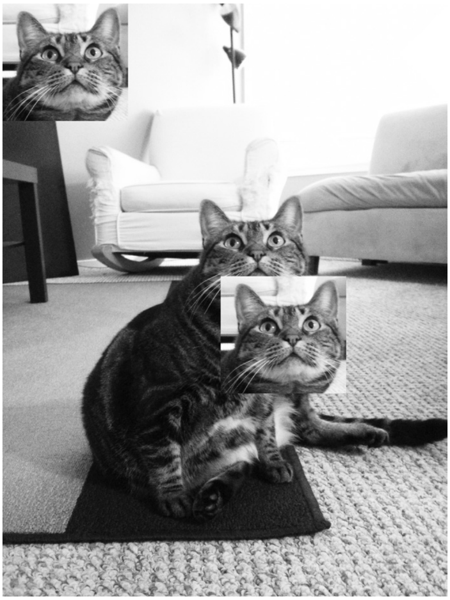


Figure 21-5: Zophie the cat, with her face pasted twice

Say you want to tile Zophie's head across the entire image, as in Figure 21-6.

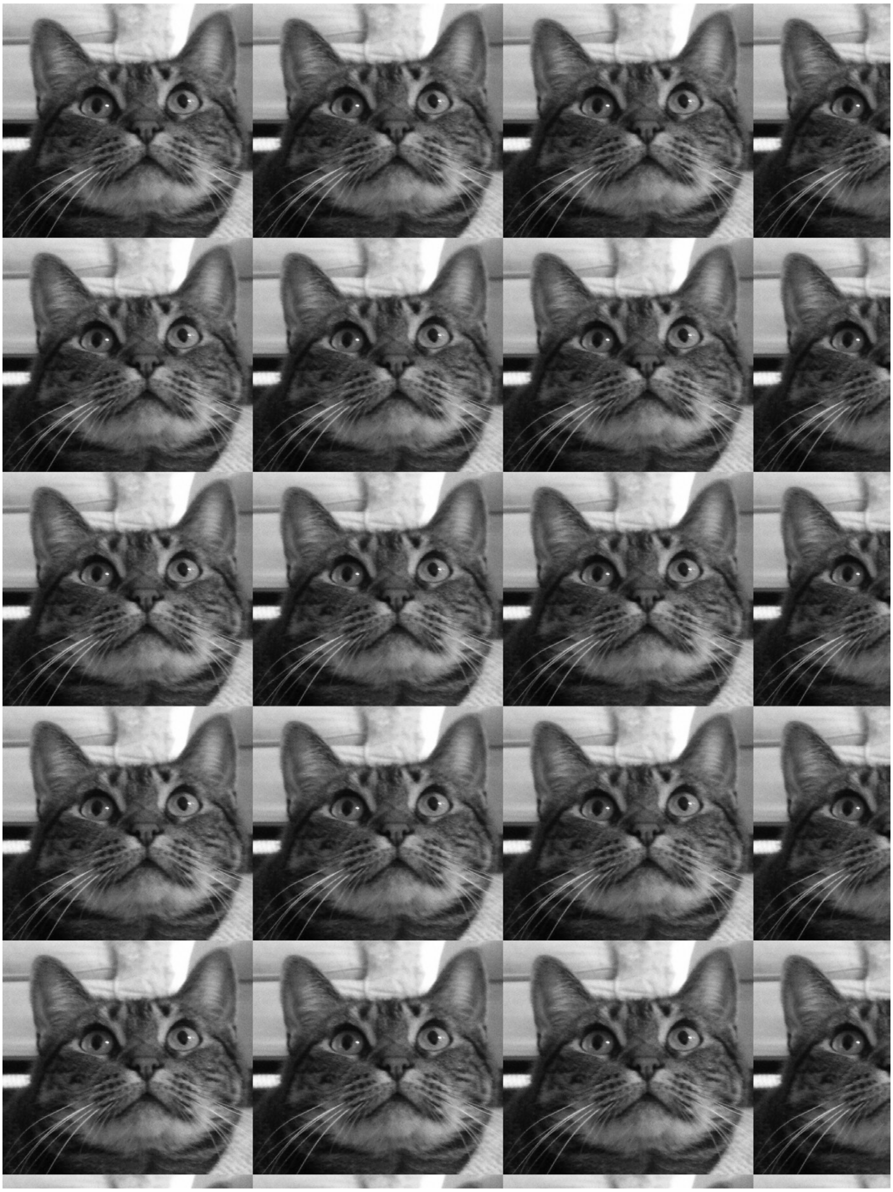


Figure 21-6: Nested `for` *loops used with* `paste()` *can duplicate the cat's face (creating a dupli-cat, if you will).*

You can achieve this effect with a couple of `for` loops. Continue the interactive shell example by entering the following:

```
  >>> cat_im_width, cat_im_height = cat_im.size
  >>> face_im_width, face_im_height = face_im.size
❶ >>> cat_copy_im = cat_im.copy()
❷ >>> for left in range(0, cat_im_width, face_im_width):
  ...   ❸ for top in range(0, cat_im_height, face_im_height):
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
...         print(left, top)
...         cat_copy_im.paste(face_im, (left, top))
...
0 0
0 215
0 430
0 645
0 860
0 1075
230 0
230 215
--snip--
690 860
690 1075
>>> cat_copy_im.save('tiled.png')
```

We store the original image's width and height in `cat_im_width` and `cat_im_height`. Next, we make a copy of the image and store it in `cat_copy_im` ❶. Now we can loop, pasting `face_im` onto the copy. The outer `for` loop's `left` variable starts at `0` and increases by `face_im_width` ❷. The inner `for` loop's `top` variable starts at `0` and increases by `face_im_height` ❸. These nested `for` loops produce values for `left` and `top` that paste a grid of `face_im` images over the `Image` object, as in Figure 21-6. To see the nested loops at work, we print `left` and `top`. After the pasting is complete, we save the modified `cat_copy_im` to *tiled.png*.

If you're pasting an image with transparency, you'll need to pass the image as the optional third argument, which tells Pillow what parts of the original image to paste. Otherwise, transparent pixels in the original image will appear as white pixels in the pasted-on image. We'll explore this practice in more detail in "Project 16: Add a Logo" on page XX.

Resizing Images

When called on an `Image` object, the `resize()` method returns a new `Image` object of the specified width and height. It accepts a two-integer tuple argument representing the new dimensions. Enter the following into the interactive shell:

```
  >>> from PIL import Image
  >>> cat_im = Image.open('zophie.png')
❶ >>> width, height = cat_im.size
❷ >>> quarter_sized_im = cat_im.resize((int(width / 2), int(height / 2)))
  >>> quarter_sized_im.save('quartersized.png')
❸ >>> svelte_im = cat_im.resize((width, height + 300))
  >>> svelte_im.save('svelte.png')
```

We assign the two values in the `cat_im.size` tuple to the variables `width` and `height` ❶. Using these variables instead of `cat_im.size[0]` and `cat_im.size[1]` makes the rest of the code more readable.

The first `resize()` call passes `int(width / 2)` for the new width and `int(height / 2)` for the new height ❷, so the `Image` object returned from `resize()` will be half the width and height of the original image, or one-quarter of the original image size overall. The `resize()` method accepts only integers in its tuple argument, which is why you needed to wrap both divisions by `2` in an `int()` call.

This resizing keeps the same proportions as the original image, but the new width and height values don't have to conserve those proportions. The `svelte_im` variable contains an `Image` object that has the original width but a height that is 300 pixels taller ❸, giving Zophie a more slender look.

Note that the `resize()` method doesn't edit the `Image` object in place but instead returns a new `Image` object.

Rotating and Flipping Images

To rotate images, use the `rotate()` method, which returns a new `Image` object and leaves the original unchanged. The method accepts a single integer or float representing the number of degrees by which to rotate the image counterclockwise. Enter the following into the interactive shell:

```
>>> from PIL import Image
>>> cat_im = Image.open('zophie.png')
>>> cat_im.rotate(90).save('rotated90.png')
>>> cat_im.rotate(180).save('rotated180.png')
>>> cat_im.rotate(270).save('rotated270.png')
```

Note that you can chain method calls by calling `save()` directly on the `Image` object returned from `rotate()`. The first `rotate()` and `save()` chain rotates the image counterclockwise by 90 degrees and saves it to *rotated90.png*. The second and third calls do the same, except they rotate the image by 180 degrees and 270 degrees, respectively. The results look like Figure 21-7.

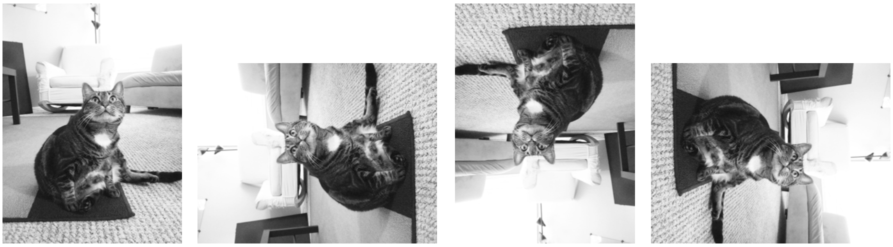


Figure 21-7: The original image (left) and the image rotated counterclockwise by 90, 180, and 270 degrees

The rotated images will have the same height and width as the original image. On Windows, a black background will fill in any gaps made by the rotation, as in Figure 21-8. On macOS and Linux, transparent pixels will fill in the gaps instead.

The `rotate()` method has an optional `expand` keyword argument that can be set to `True` to enlarge the dimensions of the image to fit the entire rotated new image. For example, enter the following into the interactive shell:

```
>>> cat_im.rotate(6).save('rotated6.png')
>>> cat_im.rotate(6, expand=True).save('rotated6_expanded.png')
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The first call rotates the image by six degrees and saves it to *rotated6.png*. (See the image on the left of Figure 21-8.) The second call rotates the image by six degrees, sets `expand` to `True`, and saves the image to *rotated6_expanded.png*. (See the image on the right of Figure 21-8.)

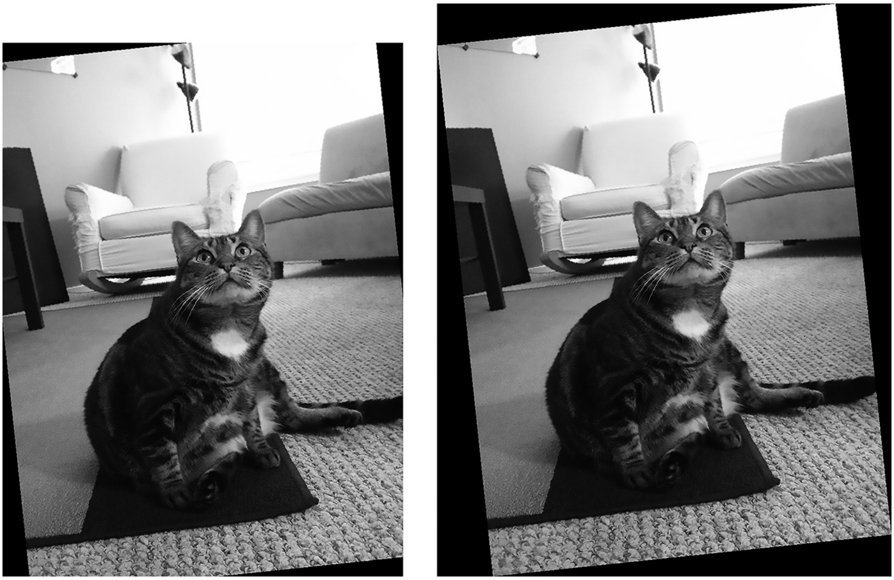


Figure 21-8: The image rotated by six degrees normally (left) and with `expand=True` (right)

If you rotate the image by 90, 180, or 270 degrees with `expand=True`, the rotated image won't have a black or transparent background.

You can also mirror-flip an image, as in Figure 21-9, with the `transpose()` method.

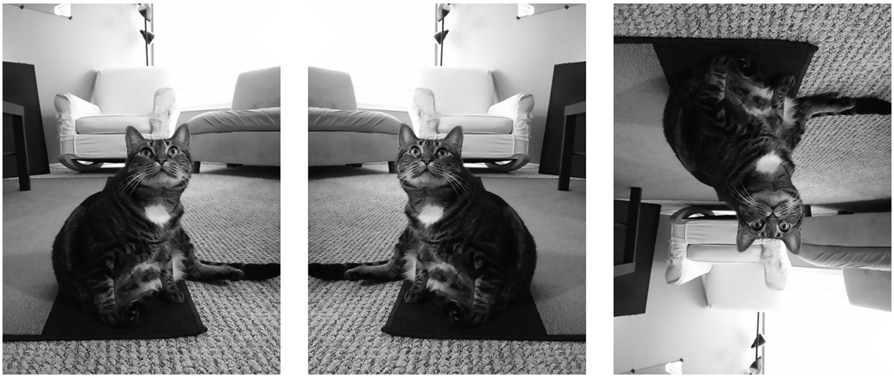


Figure 21-9: The original image (left), the image with a horizontal flip (center), and the image with a vertical flip (right)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Enter the following into the interactive shell:

```
>>> cat_im.transpose(Image.FLIP_LEFT_RIGHT).save('horizontal_flip.png')
>>> cat_im.transpose(Image.FLIP_TOP_BOTTOM).save('vertical_flip.png')
```

Like `rotate()`, `transpose()` creates a new `Image` object. We pass `Image.FLIP_LEFT_RIGHT` to flip the image horizontally and then save the result to *horizontal_flip.png*. To flip the image vertically, we pass `Image.FLIP_TOP_BOTTOM` and save the result to *vertical_flip.png*.

Changing Individual Pixels

The `getpixel()` method can retrieve the color of an individual pixel, and `putpixel()` can additionally alter that color. Both methods accept a tuple representing the pixel's x- and y-coordinates. The `putpixel()` method takes an additional argument for the pixel's new color, either as a four-integer RGBA tuple or as a three-integer RGB tuple. Enter the following into the interactive shell:

```
  >>> from PIL import Image
❶ >>> im = Image.new('RGBA', (100, 100))
❷ >>> im.getpixel((0, 0))
  (0, 0, 0, 0)
❸ >>> for x in range(100):
  ...     for y in range(50):
  ...       ❹ im.putpixel((x, y), (210, 210, 210))
  ...
  >>> from PIL import ImageColor
❺ >>> for x in range(100):
  ...     for y in range(50, 100):
  ...       ❻ im.putpixel((x, y), ImageColor.getcolor('darkgray', 'RGBA'))
  ...
  >>> im.getpixel((0, 0))
  (210, 210, 210, 255)
  >>> im.getpixel((0, 50))
  (169, 169, 169, 255)
  >>> im.save('putPixel.png')
```

We make a new image that is a 100 × 100 transparent square ❶. Calling `getpixel()` on coordinates in this image returns `(0, 0, 0, 0)` because the image is transparent ❷. To color its pixels, we use nested `for` loops to cycle through the pixels in the top half of the image ❸, passing `putpixel()` an RGB tuple representing a light gray ❹.

Say we want to color the bottom half of the image dark gray but don't know the RGB tuple for dark gray. The `putpixel()` method doesn't accept a standard color name like `'darkgray'`, so we use `ImageColor.getcolor()` to get a corresponding color tuple ❻. We loop through the pixels in the bottom half of the image ❺ and pass `putpixel()` the return value of this call, producing an image that is half light gray and half dark gray, as shown in Figure 21-10. We call `getpixel()` on any of the coordinates to confirm that the color of a given pixel is what we expect. Finally, we save the image to *putPixel.png*.

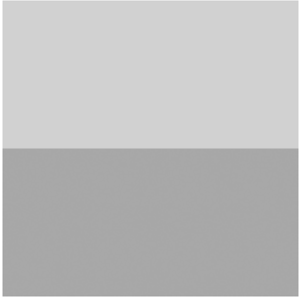


Figure 21-10: The putPixel.png *image*

Of course, drawing one pixel of an image at a time isn't very convenient. If you need to draw shapes, use the `ImageDraw` functions explained in "Drawing on Images" on page XX.

Project 16: Add a Logo

Say you have the boring job of resizing thousands of images and adding a small logo watermark to the corner of each. Doing this with a basic graphics program such as Paintbrush or Paint would take forever. A fancier graphics application such as Photoshop can do batch processing, but that software costs hundreds of dollars. Let's write a script to do it instead.

Imagine that Figure 21-11 is the logo you want to add to the bottom-right corner of each image: a black cat icon with a white border and transparent background. You can use your own logo image or download the one included in this book's online resources.

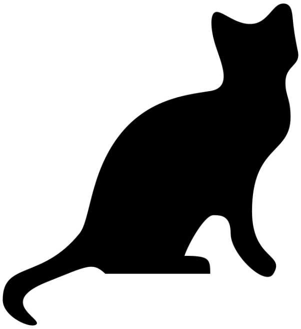


Figure 21-11: The logo to add to the image

At a high level, here's what the program should do:

- Load the logo image.
- Loop over all *.png* and *.jpg* files in the working directory.
- Check whether the image is wider and taller than 300 pixels.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

- If so, reduce the width or height (whichever is larger) to 300 pixels and scale down the other dimension proportionally.
- Paste the logo image into the corner.
- Save the altered images to another folder.

This means the code will need to do the following:

- Open the *catlogo.png* file as an `Image` object.
- Loop over the strings returned from `os.listdir('.')`.
- Get the width and height of the image from the `size` attribute.
- Calculate the new width and height of the resized image.
- Call the `resize()` method to resize the image.
- Call the `paste()` method to paste the logo in the bottom-right corner.
- Call the `save()` method to save the changes, using the original filename.

Step 1: Open the Logo Image

Open a new file editor tab, enter the following code, and save it as *resizeAndAddLogo.py*:

```
# Resizes images to fit in a 300x300 square with a logo in the corner
import os
from PIL import Image

❶ SQUARE_FIT_SIZE = 300
❷ LOGO_FILENAME = 'catlogo.png'

❸ logo_im = Image.open(LOGO_FILENAME)
❹ logo_width, logo_height = logo_im.size

# TODO: Loop over all files in the working directory.

# TODO: Check if the image needs to be resized.

# TODO: Calculate the new width and height to resize to.

# TODO: Resize the image.

# TODO: Add the logo.

# TODO: Save changes.
```

By setting up the `SQUARE_FIT_SIZE` ❶ and `LOGO_FILENAME` ❷ constants at the start of the program, we've made it easy to change the program later. Say the logo that you're adding isn't the cat icon, or say you're reducing the output images' largest dimension to something other than 300 pixels. You can straightforwardly open the code and change those values once. You could also set the values of these constants by accepting command line arguments. Without these constants, you'd instead have to search the code for all instances of `300` and `'catlogo.png'` and replace them with the new values.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The `Image.open()` method returns the logo `Image` object ❸. For readability, we assign the logo's width and height to variables ❹. The rest of the program is a skeleton of `TODO` comments.

Step 2: Loop Over All Files

Now you need to find every *.png* file and *.jpg* file in the current working directory. You don't want to add the logo image to the logo image itself, so the program should skip any image with a filename that is the same as `LOGO_FILENAME`. Add the following to your code:

```
# Resizes images to fit in a 300x300 square with a logo in the corner
import os
from PIL import Image

--snip--

os.makedirs('withLogo', exist_ok=True)
# Loop over all files in the working directory.
❶ for filename in os.listdir('.'):
    ❷ if not (filename.endswith('.png') or filename.endswith('.jpg')) \
           or filename == LOGO_FILENAME:
          ❸ continue  # Skip non-image files and the logo file itself.

    ❹ im = Image.open(filename)
      width, height = im.size

--snip--
```

First, the `os.makedirs()` call creates a *withLogo* folder in which to store the modified images, rather than overwriting the original image files. The `exist_ok=True` keyword argument will keep `os.makedirs()` from raising an exception if *withLogo* already exists. While the code loops through all the files in the working directory ❶, a long `if` statement checks for filenames that don't end with *.png* or *.jpg* ❷. If it finds any—or if the file is the logo image itself—the loop should skip it and use `continue` to go to the next file ❸. If `filename` does end with `'.png'` or `'.jpg'` and isn't the logo file, the code opens it as an `Image` object ❹ and saves its width and height.

Step 3: Resize the Images

The program should resize the image only if the width or height is larger than `SQUARE_FIT_SIZE` (300 pixels, in this case), so you should put the resizing code inside an `if` statement that checks the `width` and `height` variables. Add the following code to your program:

```
# Resizes images to fit in a 300x300 square with a logo in the corner
import os
from PIL import Image

--snip--
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

```
    # Check if the image needs to be resized.
    if width > SQUARE_FIT_SIZE and height > SQUARE_FIT_SIZE:
        # Calculate the new width and height to resize to.
        if width > height:
          ❶ height = int((SQUARE_FIT_SIZE / width) * height)
            width = SQUARE_FIT_SIZE
        else:
          ❷ width = int((SQUARE_FIT_SIZE / height) * width)
            height = SQUARE_FIT_SIZE

        # Resize the image.
        print(f'Resizing {filename}...')
      ❸ im = im.resize((width, height))

--snip--
```

If the image needs resizing, you must find out whether it's a wide or tall image. If width is greater than height, the code should reduce the height by the same proportion as the width ❶. This proportion is the SQUARE_FIT_SIZE value divided by the current width, so the code sets the new height value to this proportion multiplied by the current height value. Because the division operator returns a float value, and resize() requires the dimensions to be integers, you must remember to convert the result to an integer with the int() function. Finally, the code will set the new width value to SQUARE_FIT_SIZE.

If the height is greater than or equal to the width, the else clause performs the same calculation, but swaps the height and width variables ❷. Once those variables contain the new image dimensions, the code passes them to the resize() method and stores the returned Image object ❸.

Step 4: Add the Logo and Save the Changes

Whether or not you resized the image, you should paste the logo to its bottom-right corner. Where exactly to insert the logo depends on the size of both the image and the logo. Figure 21-12 shows how to calculate the pasting position. The left coordinate at which to paste the logo is the image width minus the logo width, and the top coordinate at which to paste the logo is the image height minus the logo height.

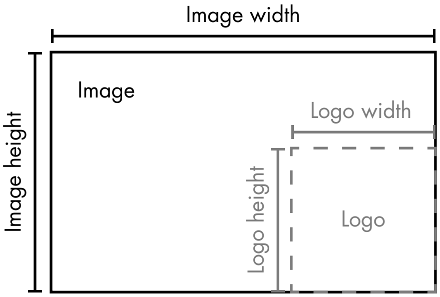


Figure 21-12: The left and top coordinates of the logo are the image width/height minus the logo width/height.

After your code pastes the logo into the image, it should save the modified `Image` object. Add the following to your program:

```
# Resizes images to fit in a 300x300 square with a logo in the corner
import os
from PIL import Image

--snip--

    # Check if the image needs to be resized.
    --snip--

    # Add the logo.
  ❶ print(f'Adding logo to {filename}...')
  ❷ im.paste(logo_im, (width - logo_width, height - logo_height), logo_im)

    # Save changes.
  ❸ im.save(os.path.join('withLogo', filename))
```

The new code prints a message telling the user that the logo is being added ❶, pastes `logo_im` onto `im` at the calculated coordinates ❷, and saves the changes to a filename in the *withLogo* directory ❸. When you run this program with the *zophie.png* and other image files in the working directory, the output will look like this:

```
Resizing zophie.png...
Adding logo to zophie.png...
Resizing zophie_xmas_tree.png...
Adding logo to zophie_xmas_tree.png...
Resizing me_and_zophie.png...
Adding logo to me_and_zophie.png...
```

The program converts *zophie.png* to a 225 × 300-pixel image that looks like Figure 21-13.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

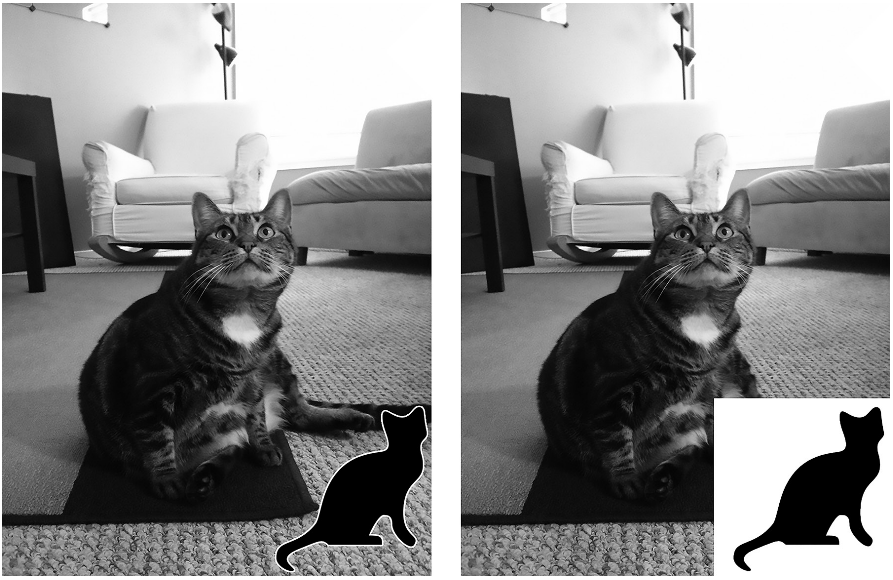


Figure 21-13: The program resized zophie.png *and added the logo (left). If you forget the third argument, the transparent pixels in the logo will appear as solid white pixels (right).*

Remember that the `paste()` method won't paste the transparency pixels unless you pass `logo_im` as the third argument. This program can automatically resize and "logo-ify" hundreds of images in just a couple of minutes.

Ideas for Similar Programs

The ability to build composite images or modify image sizes in a batch is useful for many applications. You could write similar programs to do the following:

- Add text or a website URL to images.
- Add timestamps to images.
- Copy or move images into different folders based on their sizes.
- Add a mostly transparent watermark to an image to prevent others from copying it.

Drawing on Images

If you need to draw lines, rectangles, circles, or other simple shapes on an image, use Pillow's `ImageDraw` module. For example, enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
```

First, we import `Image` and `ImageDraw`. Then, we create a 200 × 200 white image and store it in `im`. We pass this `Image` object to the `ImageDraw.Draw()` function to receive an `ImageDraw` object. This object has several methods for drawing shapes and text. Store the new object in a variable like `draw` so that you can easily use it in the following example.

Shapes

The following `ImageDraw` methods draw various kinds of shapes on the image. The `fill` and `outline` parameters for these methods are optional and will default to white if left unspecified.

Points

The `point(`*`xy, fill`*`)` method draws individual pixels. The *`xy`* argument represents a list of the points to draw. The list can contain x- and y-coordinate tuples, such as `[(x, y), (x, y), ...]`, or x- and y-coordinates without tuples, such as `[x1, y1, x2, y2, ...]`. The *`fill`* argument colors the points and can be either an RGBA tuple or a string, such as `'red'`. The *`fill`* argument is optional. The "point" name here refers to a pixel, not the unit of font size.

Lines

The `line(`*`xy, fill, width`*`)` method draws a line or series of lines. The *`xy`* argument is either a list of tuples, such as `[(x, y), (x, y), ...]`, or a list of integers, such as `[x1, y1, x2, y2, ...]`. Each point is a connecting points on the lines you're drawing. The optional *`fill`* argument specifies the color of the lines as an RGBA tuple or color name. The optional *`width`* argument determines the width of the lines, and defaults to `1` if left unspecified.

Rectangles

The `rectangle(`*`xy, fill, outline, width`*`)` method draws a rectangle. The *`xy`* argument is a box tuple of the form (*`left`*, *`top`*, *`right`*, *`bottom`*). The *`left`* and *`top`* values specify the x- and y-coordinates of the upper-left corner of the rectangle, while *`right`* and *`bottom`* specify the coordinates of the lower-right corner. The optional *`fill`* argument is the color of the inside of the rectangle. The optional *`outline`* argument is the color of the rectangle's outline. The optional *`width`* argument represents the width of the lines, and defaults to `1` if left unspecified.

Ellipses

The `ellipse(`*`xy, fill, outline, width`*`)` method draws an ellipse. If the width and height of the ellipse are identical, this method will draw a circle. The *`xy`* argument is a box tuple (*`left`*, *`top`*, *`right`*, *`bottom`*) representing a box that precisely contains the ellipse. The optional *`fill`* argument is the color of the inside of the ellipse, and the optional *`outline`* argument is the color of the ellipse's outline. The optional *`width`* argument is the width of the lines, and defaults to `1` if left unspecified.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Polygons

The `polygon(`*`xy`*`, `*`fill`*`, `*`outline`*`, `*`width`*`)` method draws an arbitrary polygon. The *`xy`* argument is a list of tuples, such as `[(x, y), (x, y), ...]`, or integers, such as `[x1, y1, x2, y2, ...]`, representing the connecting points of the polygon's sides. The last pair of coordinates will automatically connect to the first pair. The optional *`fill`* argument is the color of the inside of the polygon, and the optional *`outline`* argument is the color of the polygon's outline. The optional *`width`* argument is the width of the lines, and defaults to `1` if left unspecified.

A Drawing Example

To practice using these methods, enter the following into the interactive shell:

```
>>> from PIL import Image, ImageDraw
>>> im = Image.new('RGBA', (200, 200), 'white')
>>> draw = ImageDraw.Draw(im)
>>> draw.line([(0, 0), (199, 0), (199, 199), (0, 199), (0, 0)], fill='black') ❶
>>> draw.rectangle((20, 30, 60, 60), fill='blue') ❷
>>> draw.ellipse((120, 30, 160, 60), fill='red') ❸
>>> draw.polygon(((57, 87), (79, 62), (94, 85), (120, 90), (103, 113)), fill='brown') ❹
>>> for i in range(100, 200, 10): ❺
...         draw.line([(i, 0), (200, i - 100)], fill='green')

>>> im.save('drawing.png')
```

After making an `Image` object for a 200 × 200 white image, passing it to `ImageDraw.Draw()` to get an `ImageDraw` object, and storing the `ImageDraw` object in `draw`, we can call drawing methods on `draw`. Here, we make a thin, black outline at the edges of the image ❶, a blue rectangle whose top-left corner is at `(20, 30)` and whose bottom-right corner is at `(60, 60)` ❷, a red ellipse defined by a box from `(120, 30)` to `(160, 60)` ❸, a brown polygon with five points ❹, and a pattern of green lines drawn with a `for` loop ❺. The resulting *drawing.png* file will look like Figure 21-14 (though the colors aren't printed in this book).

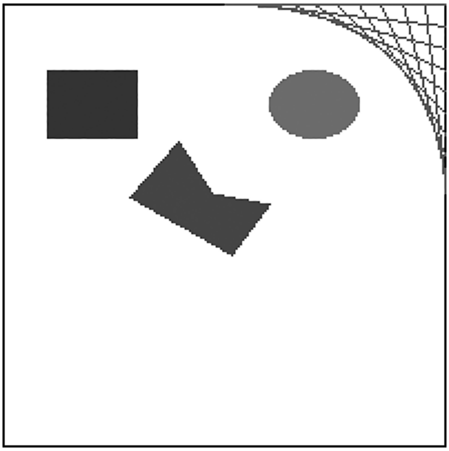


Figure 21-14: The resulting drawing.png *image*

You can use several other shape-drawing methods on `ImageDraw` objects. The full documentation is available at *https://pillow.readthedocs.io/en/latest/reference/ImageDraw.html.*

Text

The `ImageDraw` object also has a `text()` method for drawing text onto an image. This method takes four arguments:

xy A two-integer tuple specifying the upper-left corner of the text box

text The string of text you want to write

fill The color of the text

font An optional `ImageFont` object used to set the typeface and size of the text

Because it's often hard to know in advance the size of a block of text in a given font, the `ImageDraw` module offers a `textsize()` method. Its first argument is the string to measure, and its second argument is an optional `ImageFont` object. The method will return a two-integer tuple representing the width and height of the text in the given font if written onto the image. You can use this width and height to help you calculate exactly where you want to put the text on your image or find out if the text is too large for the image.

Before we use `text()` to draw text onto an image, let's discuss the optional *font* argument in more detail. This argument is an `ImageFont` object, which you can get by running the following:

```
>>> from PIL import ImageFont
```

Once you've imported Pillow's `ImageFont` module, access the font by calling the `ImageFont.truetype()` function, which takes two arguments. The first is a string representing the font's *TrueType* file, the actual font file that lives on your hard drive. A TrueType file has the *.ttf* file extension and usually

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

lives in *C:\Windows\Fonts* on Windows, */Library/Fonts* and */System/Library/Fonts* on macOS, and */usr/share/fonts/truetype* on Linux. You don't need to enter these paths as part of the TrueType file string, because Pillow knows to automatically search these directories, but it will display an error if it's unable to find the font you specified.

The second argument to `ImageFont.truetype()` is an integer for the font size in points (rather than pixels). Pillow creates PNG images that are 72 pixels per inch by default, and a point is 1/72 of an inch. For practice, enter the following into the interactive shell:

```
  >>> from PIL import Image, ImageDraw, ImageFont
  >>> import os
❶ >>> im = Image.new('RGBA', (200, 200), 'white')
❷ >>> draw = ImageDraw.Draw(im)
❸ >>> draw.text((20, 150), 'Hello', fill='purple')
❹ >>> arial_font = ImageFont.truetype('arial.ttf', 32)
❺ >>> draw.text((100, 150), 'Howdy', fill='gray', font=arial_font)
  >>> im.save('text.png')
```

After importing `Image`, `ImageDraw`, `ImageFont`, and `os`, we make an `Image` object for a new 200 × 200 white image ❶ and create an `ImageDraw` object from the `Image` object ❷. We use `text()` to write *Hello* at `(20, 150)` in purple ❸. We didn't pass the optional fourth argument in this call, so the text's typeface and size aren't customized.

Next, to set a typeface and size, we call `ImageFont.truetype()`, passing it the *.ttf* file for the desired font, followed by an integer font size ❹. We store the returned `Font` object in a variable, then pass the variable to the `text()` method's final keyword argument. The method call draws *Howdy* at `(100, 150)` in gray in 32-point Arial ❺. The resulting *text.png* file looks like Figure 21-15.

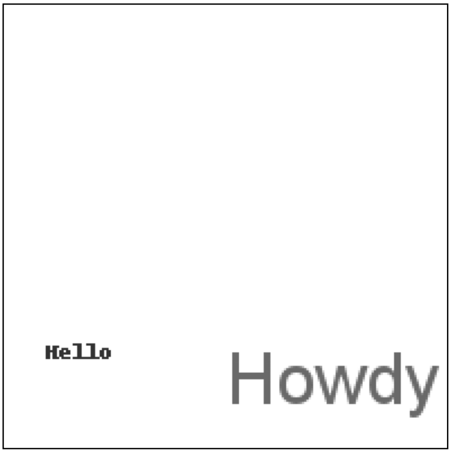


Figure 21-15: The resulting text.png *image*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If you're interested in creating computer-generated art with Python, check out *Learn Python Visually* by Tristan Bunn (No Starch Press, 2021) or my book *The Recursive Book of Recursion* (No Starch Press, 2022).

Copying and Pasting Images to the Clipboard

Just as the third-party pyperclip module allows you to copy and paste text strings to the clipboard, the pyperclipimg module can copy and paste Pillow Image objects. To install pyperclipimg, see the instructions in Appendix A.

The pyperclipimg.copy() function takes a Pillow Image object and puts it on your operating system's clipboard. You can then paste it into a graphics or image processing program such as MS Paint. The pyperclipimg.paste() function returns the image contents of the clipboard as an Image object. With *zophie.png* in the current working directory, enter the following into the interactive shell:

```
>>> from PIL import Image
>>> im = Image.open('zophie.png')
>>> import pyperclipimg
>>> pyperclipimg.copy(im)
>>> # Paste the clipboard contents to a graphics program.
>>> # Now copy a new image to the clipboard.
>>> pasted_im = pyperclipimg.paste()
>>> pasted_im.show()  # Shows the image from the clipboard.
```

In this code, we first open the *zophie.png* image as an Image object, then pass it to pyperclipimg.copy() to copy it to the clipboard. You can verify that the copy worked by pasting the image into a graphics program. Next, copy a new image from a graphics program or by right-clicking an image in your web browser and copying it. Calling pyperclipimg.paste() returns this image as an Image object in the pasted_im variable. You can verify that the paste worked by viewing it with pasted_im.show().

The pyperclipimg module can be useful as a way to let users input and output image data to your Python programs.

Creating Graphs with Matplotlib

Drawing your own graphs using Pillow is possible but would require a lot of work. The Matplotlib library creates a wide variety of graphs for use in professional publications. In this chapter, we'll create basic line graphs, bar graphs, scatter plots, and pie charts, but Matplotlib is able to create more complex 3D graphs as well. You can find the full documentation at *https://matplotlib.org*. Install Matplotlib by following the instructions in Appendix A.

Line Graphs and Scatter Plots

Let's start by creating a 2D line graph with two axes, x and y. A line graph is ideal for showing changes in one measure over time. In Matplotlib, the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

terms *plot*, *graph*, and *chart* are often used interchangeably, and the term *figure* refers to the window that contains one or more plots. Enter the following into the interactive shell:

```
>>> import matplotlib.pyplot as plt ❶
>>> x_values = [0, 1, 2, 3, 4, 5]
>>> y_values1 = [10, 13, 15, 18, 16, 20]
>>> y_values2 = [9, 11, 18, 16, 17, 19]
>>> plt.plot(x_values, y_values1) ❷
[<matplotlib.lines.Line2D object at 0x000002501D9A7D10>]>>> plt.plot(x_values, y_values2)
[<matplotlib.lines.Line2D object at 0x00000250212AC6D0>]
>>> plt.savefig('linegraph.png')  # Saves the plot as an image file
>>> plt.show()  # Opens a window with the plot
>>> plt.show()  # Does nothing
```

We import `matplotlib.pyplot` under the name `plt` ❶ to make it easier to enter its functions. Next, to plot data points to a 2D figure, we must call the `plt.plot()` function. We first save a list of integers or floats in `x_values` for the x-axis, and then save a list of integers or floats in `y_values1` for the y-axis ❷. The first values in the x-axis and y-axis lists are associated with each other, the second values in the two lists are associated with each other, and so on. After calling `plt.plot()` with these values, we call it a second time with `x_values` and `y_values2` to add a second line to the graph.

Matplotlib will automatically select colors for the lines and an appropriate size for the graph. We can save the default graph as a PNG image by calling `plt.savefig('linegraph.png')`.

Matplotlib has a preview feature that shows you the graph in a window, much like Pillow has the `show()` method for previewing `Image` objects. Call `plt.show()` to open the graph in a window. It will look like Figure 21-16.

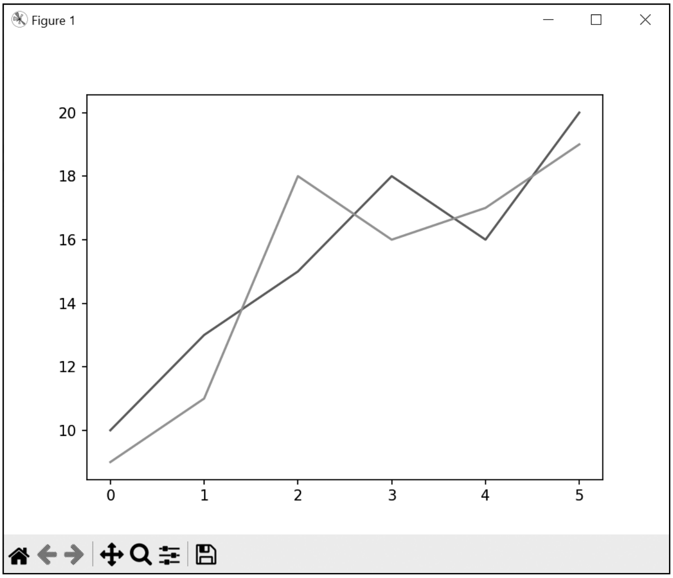


Figure 21-16: A line graph displayed with `plt.show()`

The window that plt.show() creates is interactive: you can move the graph around or zoom in or out. The house icon in the lower-left corner resets the view, and the floppy disk icon allows you to save the graph as an image file. If you're experimenting with data, plt.show() is a convenient visualization tool. The plt.show() function call will block and not return until the user closes this window.

When you close the window that the plt.show() method creates, you also reset the graph data. Calling plt.show() a second time either does nothing or displays an empty window. You'll have to call plt.plot() and any other plot-related functions again to re-create the graph. To save an image file of the graph, you must call plt.savefig() before calling plt.show().

To create a scatter plot of this same data, pass the x-axis and y-axis values to the plt.scatter() function:

```
>>> import matplotlib.pyplot as plt
>>> x_values = [0, 1, 2, 3, 4, 5]
>>> y_values1 = [10, 13, 15, 18, 16, 20]
>>> y_values2 = [9, 11, 18, 16, 17, 19]
>>> plt.scatter(x_values, y_values1)
<matplotlib.collections.PathCollection object at 0x00000250212CBAD0>
>>> plt.scatter(x_values, y_values2)
<matplotlib.collections.PathCollection object at 0x000002502132DC10>
>>> plt.savefig('scatterplot.png')
>>> plt.show()
```

When you call plt.show(), Matplotlib displays the plot in Figure 21-17. The code to create a scatter plot is identical to the code that creates a line graph, except for the function call.

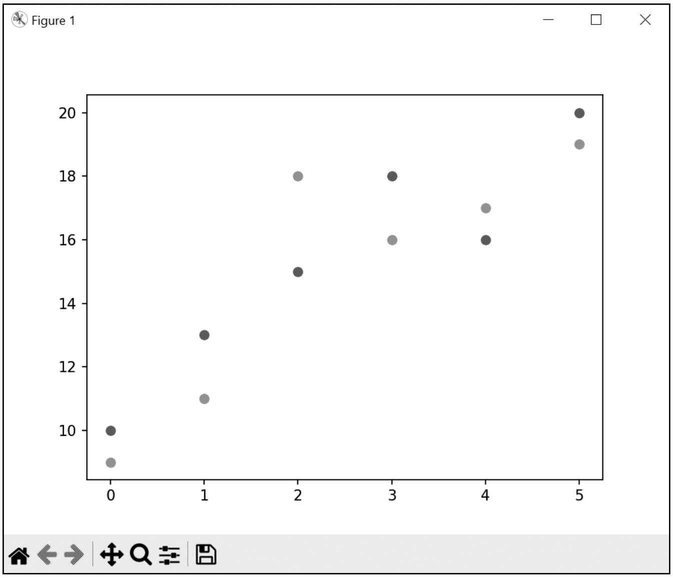


Figure 21-17: A scatter plot displayed with plt.show()

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

If you compare this graph to the line graph in Figure 21-16, you'll see the data is the same, though the scatter plot uses points instead of connected lines.

Bar Graphs and Pie Charts

Let's create a basic bar graph using Matplotlib. Bar graphs are useful for comparing the same type of data in different categories. Unlike a line graph, the order of the categories isn't important, though they're often listed alphabetically. Enter the following into the interactive shell:

```
>>> import matplotlib.pyplot as plt
>>> categories = ['Cats', 'Dogs', 'Mice', 'Moose']
>>> values = [100, 200, 300, 400]
>>> plt.bar(categories, values)
<BarContainer object of 4 artists>
>>> plt.savefig('bargraph.png')
>>> plt.show()
```

This code creates the bar graph shown in Figure 21-18. We pass the categories to list on the x-axis as the first list argument to `plt.bar()` and the values for each category as the second list argument.

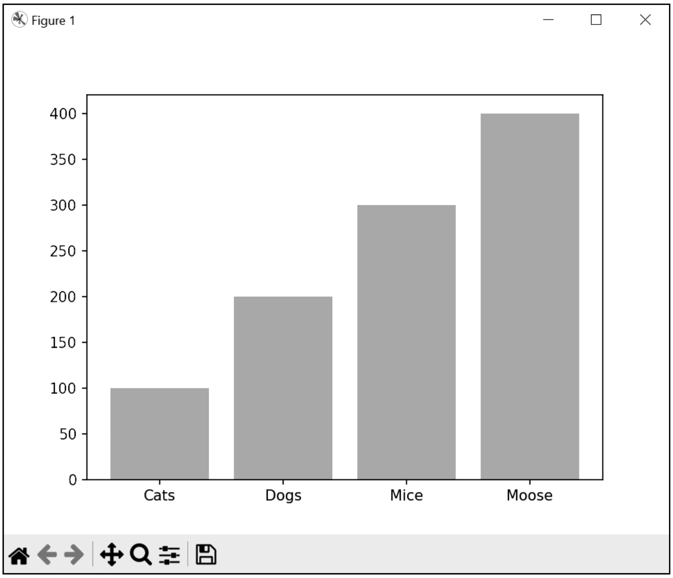


Figure 21-18: A bar graph displayed with `plt.show()`

Remember that closing the `plt.show()` window resets the graph data.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

To create a pie chart, call the `plt.pie()` function. Instead of categories and values, a pie chart has labels and slices. Enter the following into the interactive shell:

```
>>> import matplotlib.pyplot as plt
>>> slices = [100, 200, 300, 400]  # The size of each slice
>>> labels = ['Cats', 'Dogs', 'Mice', 'Moose']  # The name of each slice
>>> plt.pie(slices, labels=labels, autopct='%.1f%%')
([<matplotlib.patches.Wedge object at 0x00000218F32BA950>,
--snip--
>>> plt.savefig('piechart.png')
>>> plt.show()
```

When you call `plt.show()` for the pie chart, Matplotlib displays it in a window, like in Figure 21-19. The `plt.pie()` function accepts a list of slice sizes and a list of labels for each slice.

The `autopct` argument specifies the precision of the percentage label for each slice. The argument is a format specifier string; the `'%.1f%%'` string specifies that the number should show one digit after the decimal point. If you leave this keyword argument out of the function call, the pie chart won't list the percentage text.

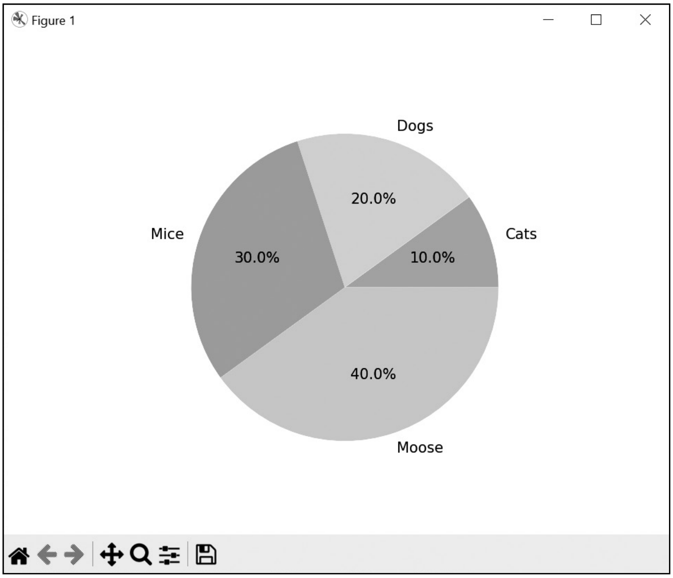


Figure 21-19: A pie chart displayed with `plt.show()`

Matplotlib automatically picks the colors for each slice, but you can customize this behavior, along with many other aspects of the graphs you create.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Additional Components

The graphs we created in the previous section are fairly basic. Matplotlib has a vast number of additional features that could fill a book of its own, so we'll look at the most common components only. Let's add data point markers, custom colors, and labels to our graphs. Enter the following into the interactive shell:

```
  >>> import matplotlib.pyplot as plt
  >>> x_values = [0, 1, 2, 3, 4, 5]
  >>> y_values1 = [10, 13, 15, 18, 16, 20]
  >>> y_values2 = [9, 11, 18, 16, 17, 19]
❶ >>> plt.plot(x_values, y_values1, marker='o', color='b', label='Line 1')
  [<matplotlib.lines.Line2D object at 0x000001BC339D2F90>]
  >>> plt.plot(x_values, y_values2, marker='s', color='r', label='Line 2')
  [<matplotlib.lines.Line2D object at 0x000001BC339D1A90>]
❷ >>> plt.legend()
  <matplotlib.legend.Legend object at 0x000001BC20915B90>
❸ >>> plt.xlabel('X-axis Label')
  Text(0.5, 0, 'X-axis Label')
  >>> plt.ylabel('Y-axis Label')
  Text(0, 0.5, 'Y-axis Label')
  >>> plt.title('Graph Title')
  Text(0.5, 1.0, 'Graph Title')
❹ >>> plt.grid(True)
  >>> plt.show()
```

After running this code, Matplotlib displays a window that looks like Figure 21-20. It contains the same line graph created previously, but we've added `marker`, `color`, and `label` keyword arguments to the `plt.plot()` function calls ❶. The marker creates a dot for each data point in the line. An `'o'` value makes the dot an O-shaped circle, while `'s'` makes it a square. The `'b'` and `'r'` color arguments set the line to blue and red, respectively. We give each line a label to use in the legend created by calling `plt.legend()` ❷.

We also create labels for the x-axis, the y-axis, and the entire graph itself by calling `plt.xlabel()`, `plt.ylabel()`, and `plt.title()` ❸, passing the label text as strings. Finally, passing `True` to `plt.grid()` ❹ enables a grid with lines along the x-axis and y-axis values.

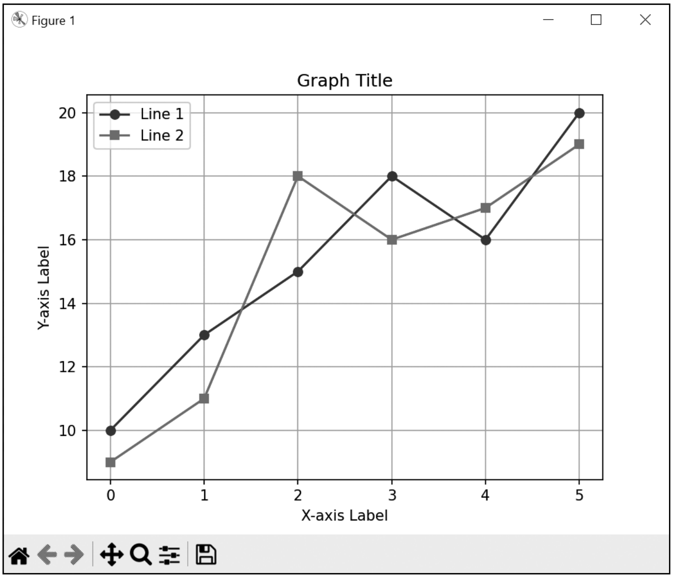


Figure 21-20: The example line graph with additional components

This is just a small sample of the features that Matplotlib provides. You can read about the other features in the online documentation.

Summary

Images consist of a collection of pixels, which each have an RGBA value for its color and a set of x- and y-coordinates representing its location. Two common image formats are JPEG and PNG. Pillow can handle both of these image formats, and others.

When a program loads an image into an `Image` object, its width and height dimensions are stored as a two-integer tuple in the `size` attribute. Objects of the `Image` data type also have methods for common image manipulations: `crop()`, `copy()`, `paste()`, `resize()`, `rotate()`, and `transpose()`. To save the `Image` object to an image file, call the `save()` method.

If you want your program to draw shapes onto an image, use `ImageDraw` methods to draw points, lines, rectangles, ellipses, and polygons. The module also provides methods for drawing text in a typeface and font size of your choosing.

While the Pillow library lets you draw shapes and individual pixels, it's easier to generate graphs using the Matplotlib library. You can create line, bar, and pie charts using Matplotlib's default settings, or you can make specific customizations. The `show()` method displays the chart on your screen for previewing, and the `save()` method generates image files you could include in documents or spreadsheets. The library's online documentation can tell you more about its rich features.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Although advanced (and expensive) applications such as Photoshop provide automatic batch processing features, you can use Python scripts to do many of the same modifications for free. In the previous chapters, you wrote Python programs to deal with plaintext files, spreadsheets, PDFs, and other formats. With Pillow, you've extended your programming powers to processing images as well!

Practice Questions

1. What is an RGBA value?
2. How can you get the RGBA value of `'CornflowerBlue'` from the Pillow module?
3. What is a box tuple?
4. What function returns an `Image` object for, say, an image file named *zophie.png*?
5. How can you find out the width and height of an `Image` object's image?
6. What method would you call to get the `Image` object for the lower-left quarter of a 100 × 100 image?
7. After making changes to an `Image` object, how could you save it as an image file?
8. What module contains Pillow's shape-drawing code?
9. `Image` objects do not have drawing methods. What kind of object does? How do you get this kind of object?
10. Which Matplotlib functions create a line graph, scatter plot, bar graph, and pie chart?
11. How can you save a Matplotlib graph as an image?
12. What does the `plt.show()` function do, and why can't you call it twice in a row?

Practice Programs

For practice, write programs to do the following tasks.

Tile Maker

Write a program that produces a tiled image from a single image, much like tiles of cat faces in Figure 21-6. Your program should have a `make_tile()` function with three arguments: a string of the image filename, an integer for how many times it should be tiled horizontally, and an integer for how many times it should be tiled vertically. The `make_tile()` function should return a larger `Image` object of the tiled image. You will use the `paste()` methods as part of this function.

For example, if *zophie_the_cat.jpg* was a 20 × 50-pixel image, calling `make_tile('zophie_the_cat.jpg', 6, 10)` should return a 120 × 500 image with

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

60 tiles total. For a bonus, try randomly flipping or rotating the image to tile when pasting it to the larger image. This tile maker works best with smaller images to tile. See what abstract art you can create with this code.

Identifying Photo Folders on the Hard Drive

I have a bad habit of transferring files from my digital camera to temporary folders somewhere on the hard drive and then forgetting about these folders. It would be nice to write a program that could scan the entire hard drive and find these leftover photo folders.

Write a program that goes through every folder on your hard drive and finds potential photo folders. Of course, first you'll have to define what you consider a "photo folder" to be; let's say that it's any folder where more than half of the files are photos. And how do you define what files are photos? First, a photo file must have the file extension *.png* or *.jpg*. Also, photos are large images; a photo file's width and height must both be larger than 500 pixels. This is a safe bet, since most digital camera photos are several thousand pixels in width and height.

As a hint, here's a rough skeleton of what this program might look like:

```
# Import modules and write comments to describe this program.

for folder_name, subfolders, filenames in os.walk('C:\\'):
    num_photo_files = 0
    num_non_photo_files = 0
    for filename in filenames:
        # Check if the file extension isn't .png or .jpg.
        if TODO:
            num_non_photo_files += 1
            continue  # Skip to the next filename.

        # Open image file using Pillow.

        # Check if the width & height are larger than 500.
        if TODO:
            # Image is large enough to be considered a photo.
            num_photo_files += 1
        else:
            # Image is too small to be a photo.
            num_non_photo_files += 1

    # If more than half of files were photos,
    # print the absolute path of the folder.
    if TODO:
        print(TODO)
```

When the program runs, it should print the absolute path of any photo folders to the screen.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Creating Custom Seating Cards

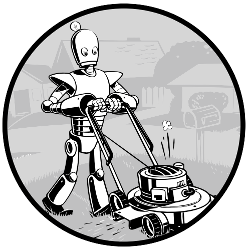
In a practice program in Chapter 17, you created custom invitations from a list of guests in a plaintext file. As an additional project, use Pillow to create images that will serve as custom seating cards for your guests. For each of the guests listed in the *guests.txt* file from the book's online resources, generate an image file with the guest's name and some flowery decoration. A public domain flower image is also available in the book's resources.

To ensure that each seating card is the same size, add a black rectangle to the edges of the invitation image; that way, when you print the image, you'll have a guideline for cutting. The PNG files that Pillow produces are set to 72 pixels per inch, so a 4 × 5-inch card would require a 288 × 360-pixel image.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

22

RECOGNIZING TEXT IN IMAGES



Text recognition, more formally called *optical character recognition (OCR)*, is the extraction of text from an image. Python has a rich collection of string methods and regular expressions for processing text, but these require you to first input the text as a string. Programs can use OCR to, for example, recognize the names on street signs and writing on checks deposited at an ATM, or to scan receipts to create electronic copies.

Like text-to-speech or speech recognition, OCR involves carrying out advanced computer science techniques, but Python modules obscure these details, making it easy to use. This chapter covers PyTesseract, the Python package that works with the open source Tesseract OCR engine. We'll also look at the free NAPS2 application, which Python can run to apply Tesseract OCR to PDF files.

Installing Tesseract and PyTesseract

To work with PyTesseract, you must install the free Tesseract OCR engine software on your Windows, macOS, or Linux computer by following the instructions in this section. You can also choose to install the language packs for non-English languages. Afterward, install the PyTesseract package so that your Python scripts can interact with Tesseract.

Windows

On Windows, open your browser to *https://github.com/UB-Mannheim/tesseract/wiki* and follow the page's instructions to download the latest installer program. Then, double-click this installer to install Tesseract.

Tesseract recognizes English text by default. During installation, you may optionally check the checkboxes for "Additional script data (download)" and "Additional language data (download)" so that Tesseract can recognize non-English letters and languages, respectively. Installing all languages adds about 600MB to the install size. These language packs have filenames identifying the language and a *.traineddata* extension, such as *jpn.traineddata* for Japanese. Alternatively, you can check the checkboxes for individual languages to save space.

After the installation has finished, add the *C:\Program Files\Tesseract-OCR* folder (or whichever folder you installed Tesseract in) to the `PATH` environment variable so that PyTesseract can access the *tesseract.exe* program. Chapter 12 covered how to modify the `PATH` environment variable.

macOS

The Homebrew package manager can install Tesseract on macOS. Navigate to *https://docs.brew.sh* to install Homebrew. Install Tesseract by opening a terminal window and running **`brew install tesseract`**, then run **`brew install tesseract-lang`** to install non-English language packs.

Linux

To install Tesseract on Linux, open a terminal window and run **`sudo apt install tesseract-ocr`**. You'll have to enter the administrator password to run this command.

To install the language packs for every language, run **`sudo apt-get install tesseract-ocr-all`** from the terminal. To install just the language packs you want, replace `all` with a three-character ISO 639 language code, such as `fra` for French, `deu` for German, or `jpn` for Japanese.

PyTesseract

After installing the Tesseract OCR engine, you can install the latest version of PyTesseract by following the instructions in Appendix A. PyTesseract also installs the Pillow image library.

OCR Fundamentals

Using PyTesseract and the Pillow image library, you can extract text from an image in four lines of code. You'll need to import the PyTesseract and Pillow libraries, open the image using the `Image.open()` function, and then finally pass the opened image to the `tess.image_to_string()` function.

Let's walk through a basic example: extracting the text from a screenshot of the introduction of my book *The Big Book of Small Python Projects* (No Starch Press, 2021). Download the *ocr-example.png* image from the book's online resources at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*, then enter the following into the interactive shell to open the image with Pillow and scan it with Tesseract:

```
>>> import pytesseract as tess
>>> from PIL import Image
>>> img = Image.open('ocr-example.png')
>>> text = tess.image_to_string(img)
>>> print(text)
This book provides you with practice examples of how programming
concepts are applied, with a collection of over 80 games, simulations, and dig-
ital art programs. These aren't code snippets; they're full, runnable Python
programs. You can copy their code to become familiar with how they work,
experiment with your own changes, and then attempt to re-create them on
your own as practice. After a while, you'll start to get ideas for your own pro-
grams and, more importantly, know how to go about creating them.
--snip--
```

Converting text from an image file into a string requires sophisticated algorithms, but Python makes these accessible with four lines of code!

Preprocessing an Image

The text in the image from the previous section extracted almost perfectly to a Python string. However, OCR has limitations. Unlike computer-generated images such as screenshots, scanned or photographed paper can contain flaws, and photographs of real-world scenes are far too complicated to extract text from. You cannot, say, take a photo of the back of a car and expect Tesseract to extract the license plate number from it. You'd first need to crop the image around the license plate; even then, it may be unreadable. For that reason, Tesseract is intended for print documents rather than photos or handwritten text.

Even in screenshots, always consider OCR text to be imperfect and in need of correction. In particular, you might encounter issues like the following:

- The string maintains any end-of-line hyphenation (for example, in "dig-" and "ital" or "pro-" and "grams").
- The string doesn't conserve any font or size information.
- The whitespace in the string may not match the text.

- The string may have incorrectly scanned characters, such as confusing lowercase *j* and lowercase *i*.
- If the image contains tables or multiple columns of text, the string may mix the text and include it out of order.

Pay special attention to mistakes in numbers, as they can be harder to spot than misspelled words.

Tesseract takes preprocessing steps to mitigate certain issues, but you can possibly improve its accuracy by using an image editing program to perform the following preprocessing steps:

- Don't scan multicolumn images; put each column of text into a separate image.
- Use only typewritten text, not handwritten text.
- Use conventional fonts, not cursive or stylized fonts.
- Rotate the image so that the lines of text are perfectly upright and not skewed at a slight angle.
- Use dark text on a light background, not white text on a black background.
- Remove any dark borders at the edges of the image.
- Add a small white border if the text runs up against the edge of the image.
- Adjust the brightness and contrast of the image so that the text stands out from the background.
- Remove small bits of "noise" pixels to clean up the image before scanning.

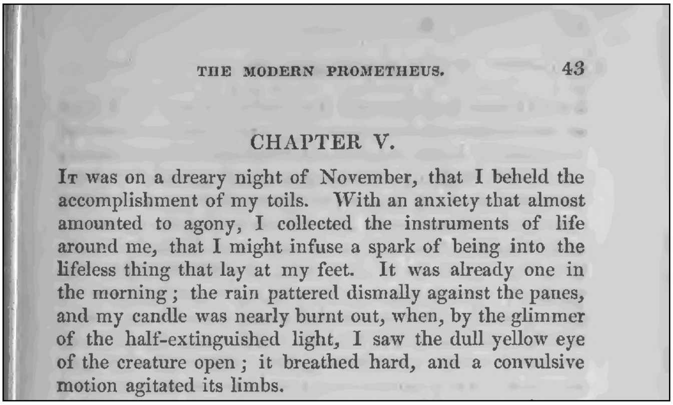
Some of these steps can be performed automatically with Python using the OpenCV library. Check out the blog post "Preprocessing Images for OCR with Python and OpenCV" at *https://autbor.com/preprocessingocr* for more examples.

Fixing Mistakes Using Large Language Models

The kinds of mistakes that OCR algorithms tend to make involve spacing and individual characters. Using a spellcheck algorithm won't find OCR errors: it will point out the accurately recognized characters of words misspelled in the original image and miss errors that result in correctly spelled words. Identifying these kinds of character mistakes requires understanding context and a common sense for what the characters should be.

This is exactly the type of problem that large language model (LLM) AIs such as ChatGPT, Gemini, and LLaMA can solve. For example, consider Figure 22-1, a raw scan from Mary Shelley's novel *Frankenstein*. This particular page was printed in 1831, so the paper is wrinkled and yellowed, with inconsistently inked characters. You can download *frankenstein.png* from the book's online resources.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart



THE MODERN PROMETHEUS. 43

CHAPTER V.

It was on a dreary night of November, that I beheld the accomplishment of my toils. With an anxiety that almost amounted to agony, I collected the instruments of life around me, that I might infuse a spark of being into the lifeless thing that lay at my feet. It was already one in the morning; the rain pattered dismally against the panes, and my candle was nearly burnt out, when, by the glimmer of the half-extinguished light, I saw the dull yellow eye of the creature open; it breathed hard, and a convulsive motion agitated its limbs.

Figure 22-1: The top part of a scanned page from an 1831 print of Frankenstein *by Mary Shelley*

Without any preprocessing, Tesseract identifies the following text:

```
>>> import pytesseract as tess
>>> from PIL import Image
>>> img = Image.open('frankenstein.png')
>>> text = tess.image_to_string(img)
>>> print(text)
THE MODERN PROMETHEUS. 43

CHAPTER V.

Iv was on a dreary night of November, that I beheld the
accomplishment of my toils. With an anxiety that almost
amounted to agony, I collected the instruments of life
around me, that I might infuse a spark of being into the
lifeless thing that lay at my feet. It was already one in
the morning ; the rain pattered dismally against the panes,
and my candle was nearly burnt out, when, by the glimmer
of the half-extinguished light, I saw the dull yellow eye
of the creature open ; it breathed hard, and a convulsive
Motion agitated its limbs.
```

It does a pretty good job, but produces the incorrectly scanned characters `'Iv'` instead of `'IT'` at the beginning of the paragraph and `'Motion'` instead of `'motion'` on the last line. (The exact results may vary as the language packs are updated.)

To fix these issues, try feeding an LLM the following sample prompt and the text output of the scanned *Frankenstein* text:

> The following is the text output from an OCR scan. Correct any spacing, missing characters, or inaccurately recognized characters from it. Do not correct spelling or grammar mistakes that exist in the original text. Put paragraphs on a single line and undo the hyphenated words that

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

> are broken across the end of a line. Only give the corrected text without explaining what OCR is or any other preface. Here is the text:
>
> THE MODERN PROMETHEUS. 43
> CHAPTER V.
> Iv was on a dreary night of November, that I beheld the . . .

When fed this prompt, the LLM produced this output:

```
THE MODERN PROMETHEUS. 43

CHAPTER V.

It was on a dreary night of November that I beheld the accomplishment of
my toils. With an anxiety that almost amounted to agony, I collected the
instruments of life around me, that I might infuse a spark of being into
the lifeless thing that lay at my feet. It was already one in the morning;
the rain pattered dismally against the panes, and my candle was nearly burnt
out, when, by the glimmer of the half-extinguished light, I saw the dull
yellow eye of the creature open; it breathed hard, and a convulsive motion
agitated its limbs.
```

The LLM corrected `'Iv'` to `'It'` and `'Motion'` to `'motion'`. It also removed the hyphenated words at the end of each line so that the newlines could be removed. This makes it easier to, say, copy and paste the text into a Word document or email. To automate this process, most online LLMs have APIs so that your programs can directly send prompts and receive responses. Unless you run an LLM on your local machine (which is beyond the scope of this book), you'll have to register for these online LLM services. This may be free or require a subscription fee.

Always remember that LLMs are prone to overconfidence. You should always verify their output. The text they return may have missed some mistakes, fixed the wrong kinds of mistakes, or even introduced new mistakes of their own. You'll still need a human to review the machine output. (And you may want a second human to review the first human's work, as humans often make mistakes to.)

Recognizing Text in Non-English Languages

Tesseract assumes the text it is scanning is English by default, but you can specify other languages as well. "Installing Tesseract and PyTesseract" on page XX has instructions for installing non-English language packs. You can see the language packs you have installed by entering the following into the interactive shell:

```
>>> import pytesseract as tess
>>> tess.get_languages()
['afr', 'amh', 'ara', 'asm', 'aze', 'aze_cyrl', 'bel', 'ben', 'bod', 'bos',
--snip--
'ton', 'tur', 'uig', 'ukr', 'urd', 'uzb', 'uzb_cyrl', 'vie', 'yid', 'yor']
```

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

The strings in this list are mostly three-character ISO 639-3 language codes, with a few exceptions. For example, while `'aze'` is the ISO 639-3 code for the Azeri language with Latin letters, the `'aze_cyrl'` string is Azeri with Cyrillic letters. Consult the Tesseract documentation for full details.

To scan images with non-English text, pass one of these string values for the `lang` keyword argument. For example, *frankenstein_jpn.png* has a Japanese translation of a section from *Frankenstein*. Download this file from the book's online resources and enter the following into the interactive shell:

```
>>> import pytesseract as tess
>>> from PIL import Image
>>> img = Image.open('frankenstein_jpn.png')
>>> text = tess.image_to_string(img, lang='jpn')
>>> print(text)
第 5 剖 私が自分の労苦の成果を目の当たりにしたのは、11 月の芝鬱な夜でした。ほとんど苦
痛に等しい不安を抱えながら、私は足元に横たわる生命のないものに存在の輝きを吹き込むこ
--snip--
だ有目、しわが寄った顔色、そしてまっすぐな黒い大と、より恐ろしいコントラストを形成した
だけでした。
```

If you use the wrong language, `image_to_string()` returns Tesseract's best guess as to what English characters the Japanese characters looked like. Of course, since the characters aren't English, the returned text will be gibberish:

```
>>> import pytesseract as tess
>>> from PIL import Image
>>> img = Image.open('frankenstein_jpn.png')
>>> text = tess.image_to
_string(img, lang='eng')
>>> print(text)
BS FABADOABOMEE AOYEVICLEDIL, 1 ADBBERKCLE, (ELA ER
WISE LW ABBA A TRA B, ALE TIRE DO EMO REVS DICED MS EKA
--snip--
```

To recognize text in multiple languages, you can combine language codes with a `'+'` character before passing it to the `image_to_string()` function's `lang` keyword argument. For example, `tess.image_to_string(img, lang='eng+jpn')` recognizes both English and Japanese characters in an image.

The NAPS2 Scanner Application

While PyTesseract is useful for extracting text from images, a common use case for OCR is to create PDF documents of scanned images with searchable text. Although there are apps to do this, they often don't offer the flexibility needed to automate the PDF generation for hundreds or thousands of images. I recommend the open source Not Another PDF Scanner 2 (NAPS2) application not just for controlling flatbed scanners but also for its ability to run Tesseract and add text to PDF documents. It is free, has straightforward features, and is available on Windows, macOS, and Linux. NAPS2 can combine several images into a PDF file with embedded text without being connected to a physical scanner. It also knows how to use Tesseract's advanced features, so it can embed the text strings at their correct location on the PDF's pages, and you can run it from a Python script.

Installing and Setting Up NAPS2

To install NAPS2, navigate to *https://www.naps2.com/download* and download the installer for your operating system. On Windows and macOS, run the downloaded installer. On Linux, download the Flatpak installer for Tesseract. Then, open a Terminal window and run **`flatpak install naps2-X.X.X-linux-x64.flatpak`** (or whatever the downloaded installer filename is) from the download folder. You may need to enter the administrator password to finish installation.

Once it's installed, you can run the NAPS2 desktop application. On Windows, you can select NAPS2 from the Start menu. On macOS, you can run NAPS2 from Spotlight. On Linux, you'll need to open a new terminal window and run `flatpak run com.naps2.Naps2`. However, this book uses NAPS2 from Python code with the `subprocess` module instead of the graphical user interface.

Running NAPS2 from Python

Python scripts can use the `subprocess` module to run the NAPS2 application with several command line arguments. When run this way, NAPS2 does not make its application window appear, which is ideal for an automation step in a Python script.

Let's use the *frankenstein.png* image once again and have NAPS2 generate a PDF with embedded OCR text. The location of the NAPS2 program is different on each operating system; the following interactive shell code shows the path for Windows:

```
>>> import subprocess
>>> naps2_path = [r'C:\Program Files\NAPS2\NAPS2.Console.exe']  # Windows
>>> proc = subprocess.run(naps2_path + ['-i', 'frankenstein.png', '-o',
'output.pdf', '--install', 'ocr-eng', '--ocrlang', 'eng', '-n', '0', '-f',
'-v'], capture_output=True)
```

On macOS, replace the line that sets the path with the following: `naps2_path = ['/Applications/NAPS2.app/Contents/MacOS/NAPS2', 'console']`. On Linux, use the following instead: `naps2_path = ['flatpak', 'run', 'com.naps2.Naps2', 'console']`.

The code creates a new file named *output.pdf* that contains a single page with the scanned image from *frankenstein.png*. However, if you open this file in a PDF application, you'll notice that you can highlight the text and copy it to the clipboard. Many PDF applications will also let you save the PDF as a *.txt* text file of the OCR text.

Let's take a look at each of the command line arguments in this example. You can change them as needed for your own purposes:

`'-i', 'frankenstein.png'` Sets the input as the *frankenstein.png* image file. See "Specifying Input" on page XX for more information on specifying multiple inputs in various formats.

`'-o', 'output.pdf'` Creates a file named *output.pdf* to hold the OCR results.

`'--install', 'ocr-eng'` Installs the English language pack for OCR. This does nothing if the language is already installed. If you want to install a different language pack, use the `ocr-` prefix with another three-letter ISO 639 language code.

`'--ocrlang', 'eng'` Sets English as the language that the OCR scan recognizes. This argument is passed directly to Tesseract's command line argument, so you could use an argument like `'eng+jpn+rus'` to specify that the image has text in English, Japanese, and Russian.

`'-n', '0'` Specifies that you want to do zero scans and not use a flatbed scanner. This prevents error messages when there's no physical flatbed scanner connected to your computer.

`'-f'` Forces NAPS2 to overwrite the *output.pdf* output file if a file with that name already exists.

`'-v'` Enables verbose mode so that status text appears as NAPS2 creates your PDF. If you want to see this status text, change the `capture_output=True` keyword argument for `subprocess.run()` to `capture_output=False`.

The online documentation for NAPS2's command line arguments is at *https://www.naps2.com/doc/command-line*. Chapter 19 covered the `subprocess` module in more detail.

Specifying Input

NAPS2 lets you import PDFs and most image formats to create a final combined PDF. The application has its own mini language for specifying multiple inputs as a single command line argument following `-i`. This can become quite complicated, but you can think of it as semicolon delimited, with Python index and slice notation.

To specify multiple files, separate them with a semicolon. For example, passing `'-i', 'cat.png;dog.png;moose.png'` creates a PDF with *cat.png* used for the first page, *dog.png* used for the second page, and *moose.png* used for the third page.

You can also specify individual pages in a PDF with syntax that is identical to Python's list slice syntax. Follow the PDF filename with square brackets containing the page number to use. As in Python, `0` represents the first page. For example, passing `'-i', 'spam.pdf[0];spam.pdf[5];eggs.pdf'` creates a PDF with page 1 of *spam.pdf*, followed by page 6 of *spam.pdf*, and then all pages of *eggs.pdf*.

You can specify a range of pages with this slice notation or use negative numbers to represent pages from the end of the PDF document. For example, passing `'-i', 'spam.pdf[0:2];eggs.pdf[-1]'` combines the first two pages of *spam.pdf* with the last page from *eggs.pdf*.

There are several more features that NAPS2 provides through its command line arguments. Check out its online documentation to learn more about them. If you find that NAPS2 isn't suitable for your needs, I also recommend the ocrmypdf package at *https://pypi.org/project/ocrmypdf/* for creating PDFs with embedded text.

Summary

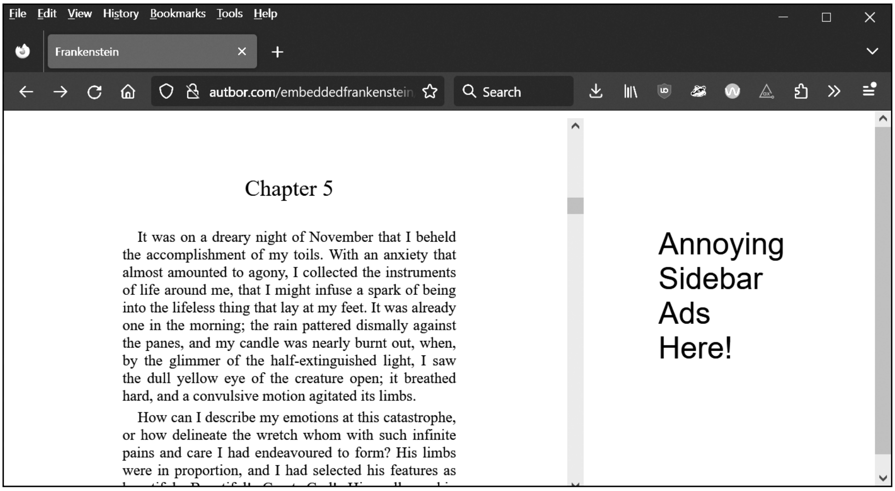
In this chapter, you learned how to harness the power of Tesseract to extract text from images. This is quite a powerful ability that can save you hours of data entry. However, OCR isn't magic, and your images may need preprocessing to get accurate results. Tesseract is also designed to work with typewritten dark text on light backgrounds where the text is level, and you must know the language of the image's text to get good results. Large language model AI can help fix incorrectly recognized characters, but its output requires human oversight as well. Finally, the open source NAPS2 application provides a way to take several images and combine them into a single PDF with embedded OCR text. OCR is an incredible breakthrough of computer science, but you don't need an advanced degree to use it. Python makes OCR accessible to everyone.

Practice Questions

1. What language does Tesseract recognize by default?
2. Name a Python image library that PyTesseract works with.
3. What PyTesseract function accepts an image object and returns a string of the text in the image?
4. If you take a photo of a street sign, will Tesseract be able to identify the sign text in the photo?
5. What function returns the list of language packs installed for Tesseract?
6. What keyword argument do you specify to PyTesseract if an image contains both English and Japanese text?
7. What application lets you create PDFs with embedded OCR text?

Practice Program: Browser Text Scraper

Some websites allow you to view their text contents but make it difficult to save or even copy and paste the text to your computer. You may see them as PDFs embedded within the web page. An example of this is at *https://autbor.com/embeddedfrankenstein/*, shown in Figure 22-2.


File Edit View History Bookmarks Tools Help
Frankenstein
autbor.com/embeddedfrankenstein
Search

Chapter 5

It was on a dreary night of November that I beheld the accomplishment of my toils. With an anxiety that almost amounted to agony, I collected the instruments of life around me, that I might infuse a spark of being into the lifeless thing that lay at my feet. It was already one in the morning; the rain pattered dismally against the panes, and my candle was nearly burnt out, when, by the glimmer of the half-extinguished light, I saw the dull yellow eye of the creature open; it breathed hard, and a convulsive motion agitated its limbs.

How can I describe my emotions at this catastrophe, or how delineate the wretch whom with such infinite pains and care I had endeavoured to form? His limbs were in proportion, and I had selected his features as

Annoying Sidebar Ads Here!

Figure 22-2: An example webpage with an embedded document

The PyAutoGUI library covered in Chapter 23 can take screenshots and save them to an image, while the Pillow library covered in Chapter 21 can crop images. PyAutoGUI also has a MouseInfo application for finding XY coordinates on the screen.

Write a program named *ocrscreen.py* that takes a screenshot, crops the image to just the text portion in the screenshot, then passes it on to PyTesseract for OCR. The program should append the recognized text to the end of a text file named *output.txt*. Here is a template for the *ocrscreen.py* program:

```
import pyautogui
# TODO - Add the additionally needed import statements.

# The coordinates for the text portion. Change as needed:
LEFT = 400
TOP = 200
RIGHT = 1000
BOTTOM = 800

# Capture a screenshot:
img = pyautogui.screenshot()

# Crop the screenshot to the text portion:
img = img.crop((LEFT, TOP, RIGHT, BOTTOM))

# Run OCR on the cropped image:
# TODO - Add the PyTesseract code here.

# Add the OCR text to the end of output.txt:
# TODO - Call open() in append mode and append the OCR text.
```

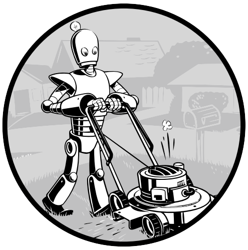
Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

This program should let you scroll the embedded, unsavable text into view in your browser, run the program, and then scroll the PDF to the next page of content. Once done, you'll have your own copy of the document's text. (If you read Chapter 23, you'll also learn how you can make your script simulate key presses to scroll the web page for you.)

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

23

CONTROLLING THE KEYBOARD AND MOUSE



Knowing various Python packages for editing spreadsheets, downloading files, and launching programs is useful, but sometimes there just aren't any packages for the applications you need to work with. The ultimate tools for automating tasks on your computer are programs that you write to directly control the keyboard and mouse. These programs can send other applications virtual keystrokes and mouse clicks, as if you were sitting at your computer and interacting with the applications yourself.

This technique is known as *graphical user interface automation,* or *GUI automation* for short. With GUI automation, your programs can do anything that a human user sitting at the computer can do, except spill coffee on the keyboard. Think of GUI automation as programming a robotic arm. You can program the robotic arm to type at your keyboard and move your mouse for you. This technique is particularly useful for tasks that involve a lot of mindless clicking or filling out of forms. This powerful technique is why account sign-up and login web pages have bot-detecting captcha challenges. Otherwise, automation programs could sign up for multiple free accounts, flood social media with spam, or guess account passwords.

Some companies sell innovative (and pricey) "automation solutions," usually marketed as *robotic process automation (RPA)* tools. These products are effectively no different from the Python scripts you can make yourself with the PyAutoGUI library, which has functions for simulating mouse movements, button clicks, and keyboard typing. This chapter covers only a subset of PyAutoGUI's features; you can find the full documentation at *https://pyautogui.readthedocs.io/*. To install the latest version of PyAutoGUI compatible with this book, follow the instructions in Appendix A.

WARNING *Don't save your programs as* pyautogui.py. *If you do, then when you run* `import pyautogui` *Python will import your program instead of PyAutoGUI, and you'll get error messages like* `AttributeError: module 'pyautogui' has no attribute 'click'`.

Setting Up Accessibility Apps on macOS

As a security measure, macOS doesn't normally let programs control the mouse or keyboard. To make PyAutoGUI work on macOS, you must set the program running your Python script to be an accessibility application. Without this step, your PyAutoGUI function calls will have no effect.

Whether you run your Python programs from Mu, IDLE, or the Terminal, keep that application open. Then, open **System Preferences** and go to the **Accessibility** tab. The currently open applications will appear under the "Allow the apps below to control your computer" label. Check Mu, IDLE, Terminal, or whichever app you use to run your Python scripts. You'll be prompted to enter your password to confirm these changes.

Staying on Track

Before you jump into a GUI automation, you should know how to escape problems that may arise. Python can move your mouse and type keystrokes at an incredible speed. In fact, it might be too fast for other programs to keep up with. Also, if something goes wrong but your program keeps moving the mouse around, it will be hard to tell exactly what the program is doing or how to recover from the problem. Like the enchanted brooms from "The Sorcerer's Apprentice" sequence in Disney's *Fantasia*, which kept filling (and then overfilling) Mickey's tub with water, your program could get out of control even though it's following your instructions perfectly. Stopping the program can be difficult if the mouse is moving around on its own, preventing you from clicking the Mu Editor window to close it.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Fortunately, there are several ways to prevent or recover from GUI automation problems.

Pauses and Fail-Safes

If your program has a bug and you're unable to use the keyboard and mouse to shut it down, you can use PyAutoGUI's fail-safe feature. Quickly slide the mouse to one of the four corners of the screen. Every PyAutoGUI function call has a one-tenth-of-a-second pause after performing its action to give you enough time to move the mouse to a corner. If PyAutoGUI then finds that the mouse cursor is in a corner, it raises the `pyautogui.FailSafeException` exception. Non-PyAutoGUI instructions won't have this pause. You can adjust this pause duration by setting `pyautogui.PAUSE` to a value other than `0.1`.

If you find yourself in a situation where you need to stop your PyAutoGUI program, just slam the mouse toward a screen corner to stop it.

Log Outs

Perhaps the simplest way to stop an out-of-control GUI automation program is to log out, which will shut down all running programs. On Windows and Linux, the log-out hotkey is CTRL-ALT-DEL. On macOS, it is ⌘-SHIFT-Q. By logging out, you'll lose any unsaved work, but at least you won't have to wait for a full reboot of the computer.

Controlling Mouse Movement

In this section, you'll learn how to move the mouse and track its position on the screen using PyAutoGUI, but first you need to understand how PyAutoGUI works with coordinates.

PyAutoGUI's mouse functions use x- and y-coordinates. Figure 23-1 shows the coordinate system for the computer screen; it's similar to the coordinate system used for images, discussed in Chapter 21. The *origin*, where *x* and *y* are both zero, is at the upper-left corner of the screen. The x-coordinates increase going to the right, and the y-coordinates increase going down. All coordinates are positive integers; there are no negative coordinates.

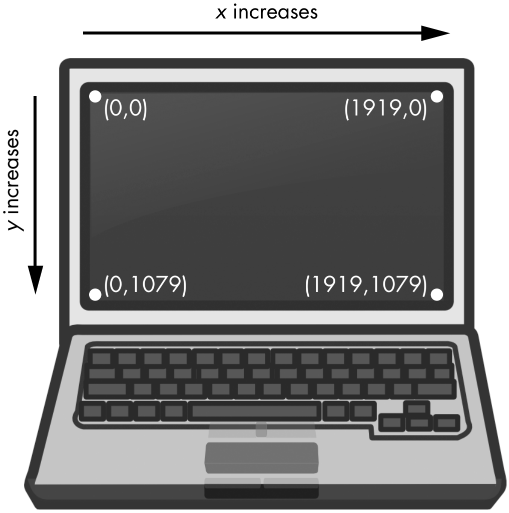


Figure 23-1: The coordinates of a computer screen with 1920 × 1080 resolution

Your *resolution* is how many pixels wide and tall your screen is. If your screen's resolution is set to 1920 × 1080, then the coordinate for the upper-left corner will be (0, 0), and the coordinate for the bottom-right corner will be (1919, 1079).

The `pyautogui.size()` function returns a `Size` named tuple of the screen's width and height in pixels. Named tuples are beyond the scope of this book, but they are basically tuples with integer indexes that also have named attributes. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> screen_size = pyautogui.size()  # Obtain the screen resolution.
>>> screen_size
Size(width=1920, height=1080)
>>> screen_size[0], screen_size[1]
(1920, 1080)
>>> screen_size.width, screen_size.height
(1920, 1080)
>>> tuple(screen_size)
(1920, 1080)
```

The `pyautogui.size()` function returns a `Size` object of `(1920, 1080)` on a computer with a 1920 × 1080 resolution; depending on your screen's resolution, your return value may be different.

Moving the Mouse

Now that you understand screen coordinates, let's move the mouse. The `pyautogui.moveTo()` function will instantly move the mouse cursor to a specified position on the screen. Integer values for the x- and y-coordinates make

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

up the function's first and second arguments, respectively. An optional duration integer or float keyword argument specifies the number of seconds it should take to move the mouse to the destination. If you leave it out, the default is 0 for instantaneous movement. (All of the duration keyword arguments in PyAutoGUI functions are optional.) Enter the following into the interactive shell:

```
>>> import pyautogui
>>> for i in range(10):  # Move the mouse in a square.
...     pyautogui.moveTo(100, 100, duration=0.25)
...     pyautogui.moveTo(200, 100, duration=0.25)
...     pyautogui.moveTo(200, 200, duration=0.25)
...     pyautogui.moveTo(100, 200, duration=0.25)
...
```

This example moves the mouse cursor clockwise in a square pattern among the four coordinates provided a total of 10 times. Each movement takes one-quarter of a second, as specified by the duration=0.25 keyword argument. If you hadn't passed a third argument to any of the pyautogui .moveTo() calls, the mouse cursor would have instantly teleported from point to point.

The pyautogui.move() function moves the mouse cursor *relative to its current position*. The following example moves the mouse in the same square pattern, except it begins the square from wherever the mouse happens to be on the screen when the code starts running:

```
>>> import pyautogui
>>> for i in range(10):
...     pyautogui.move(100, 0, duration=0.25)  # Right
...     pyautogui.move(0, 100, duration=0.25)  # Down
...     pyautogui.move(-100, 0, duration=0.25)  # Left
...     pyautogui.move(0, -100, duration=0.25)  # Up
...
```

The pyautogui.move() function also takes three arguments: how many pixels to move horizontally to the right, how many pixels to move vertically downward, and (optionally) how long it should take to complete the movement. A negative integer for the first or second argument will cause the mouse to move left or upward, respectively.

Getting the Current Position

You can determine the mouse's current position by calling the pyautogui .position() function, which will return a Point named tuple of the mouse cursor's *x* and *y* positions at the time of the function call. You can access the x- and y-coordinates either through the 0 and 1 integer indexes of the Point named tuple or through the x and y attributes. (This is similar to the

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Size named tuple's width and height attributes.) Enter the following into the interactive shell, moving the mouse around after each call:

```
>>> pyautogui.position()  # Get the current mouse position.
Point(x=311, y=622)
>>> pyautogui.position()  # Get the current mouse position again.
Point(x=377, y=481)
>>> p = pyautogui.position()  # And again
>>> p
Point(x=1536, y=637)
>>> p[0]  # The x-coordinate is at index 0.
1536
>>> p.x  # The x-coordinate is also in the x attribute.
1536
```

Of course, your return values will vary depending on where your mouse cursor is.

Controlling Mouse Interaction

Now that you know how to move the mouse and figure out where it is on the screen, you're ready to start clicking, dragging, and scrolling.

Clicking

To send a virtual mouse click to your computer, call the pyautogui.click() method. By default, this click uses the left mouse button and takes place wherever the mouse cursor is currently located. You can pass x- and y-coordinates of the click as optional first and second arguments if you want it to take place somewhere other than the mouse's current position.

If you want to specify which mouse button to use, include the button keyword argument, with a value of 'left', 'middle', or 'right'. For example, pyautogui.click(100, 150, button='left') will click the left mouse button at the coordinates (100, 150), while pyautogui.click(200, 250, button='right') will perform a right-click at (200, 250).

Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.click(10, 5)  # Move the mouse to (10, 5) and click.
```

You should see the mouse pointer move to near the top-left corner of your screen and click once. A full "click" is defined as pushing a mouse button down and then releasing it without moving the cursor. You can also perform a click by calling pyautogui.mouseDown(), which only pushes the mouse button down, and pyautogui.mouseUp(), which only releases the button. These functions have the same arguments as click(), and in fact, the click() function is just a convenient wrapper around these two function calls.

As a further convenience, the pyautogui.doubleClick() function will perform two clicks with the left mouse button. The pyautogui.rightClick() and pyautogui.middleClick() functions will perform a click with the right and middle mouse buttons, respectively.

Dragging

Dragging means moving the mouse while holding down one of the mouse buttons. For example, you can move files between folders by dragging the folder icons, or you can move appointments around in a calendar app.

PyAutoGUI provides the pyautogui.dragTo() and pyautogui.drag() functions to drag the mouse cursor to a new location or a location relative to its current one. The arguments for dragTo() and drag() are the same as moveTo() and move(): the x-coordinate/horizontal movement, the y-coordinate/vertical movement, and an optional duration of time. (The macOS operating system doesn't drag correctly when the mouse moves too quickly, so passing a duration keyword argument is recommended.)

To try these functions, open a graphics drawing application such as MS Paint on Windows, Paintbrush on macOS, or GNU Paint on Linux. (If you don't have a drawing application, you can use the online one at *https://sumopaint.com*.) I will use PyAutoGUI to draw in these applications.

With the mouse cursor over the drawing application's canvas and the Pencil or Brush tool selected, enter the following into a new file editor window and save it as *spiralDraw.py*:

```
  import pyautogui
❶ pyautogui.sleep(5)
❷ pyautogui.click()  # Click to make the window active.
  distance = 300
  change = 20
  while distance > 0:
    ❸ pyautogui.drag(distance, 0, duration=0.2)  # Move right.
    ❹ distance = distance - change
    ❺ pyautogui.drag(0, distance, duration=0.2)   # Move down.
    ❻ pyautogui.drag(-distance, 0, duration=0.2)  # Move left.
      distance = distance - change
      pyautogui.drag(0, -distance, duration=0.2)  # Move up.
```

When you run this program, there will be a five-second delay ❶ during which you can move the mouse cursor over the drawing program's window with the Pencil or Brush tool selected. PyAutoGUI's sleep() function is identical to time.sleep() but exists so that you don't need to add import time to your code. Then, *spiralDraw.py* will take control of the mouse and click to make the drawing program's window active ❷. The *active window* is the window that currently accepts keyboard input, and the actions you take (like typing or, in this case, dragging the mouse) will affect that window. The active window is also known as the *focused* or *foreground window*. Once the drawing program is active, *spiralDraw.py* draws a square spiral pattern like the one on the left of Figure 23-2.

The distance variable starts at 300, so on the first iteration of the while loop, the first drag() call drags the cursor 300 pixels to the right, taking 0.2 seconds ❸. Then, distance is decreased to 280 ❹, and the second drag()

call drags the cursor 280 pixels down ❺. The third `drag()` call drags the cursor -280 horizontally (280 to the left) ❻, `distance` is decreased to 260, and the last `drag()` call drags the cursor 260 pixels up. On each iteration, the mouse is dragged right, down, left, and up, and `distance` is slightly smaller than it was in the previous iteration. By looping over this code, you can move the mouse cursor to draw a square spiral.

While you can also create a square spiral image by using the Pillow package discussed in Chapter 21, creating the image by controlling the mouse to draw it in MS Paint lets you make use of this program's various brush styles, as shown in Figure 23-2 on the right, as well as other advanced features, such as gradients or the fill bucket. You can preselect the brush settings yourself (or have your Python code select these settings) and then run the spiral-drawing program.

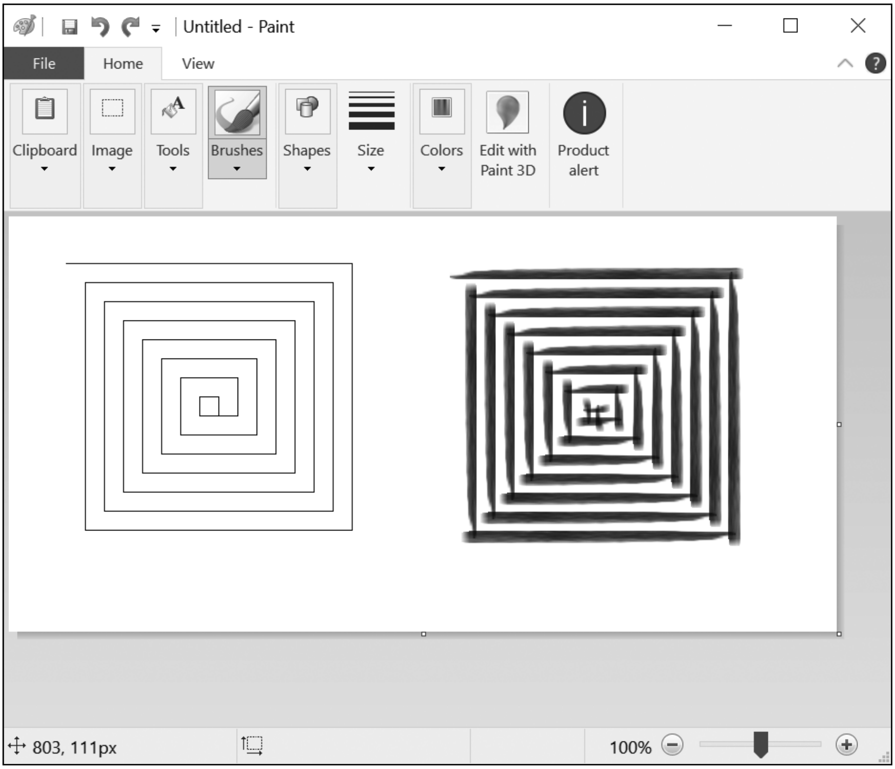


Figure 23-2: The results from the `pyautogui.drag()` *example, drawn with MS Paint's different brushes*

You could draw this spiral by hand (or rather, by mouse), but you'd have to work slowly to be so precise. PyAutoGUI can do it in a few seconds!

Scrolling

The final PyAutoGUI mouse function is `scroll()`, to which you pass an integer argument for how many units you want to scroll the mouse up or down.

The size of a unit varies for each operating system and application, so you'll have to experiment to see exactly how far it scrolls in your particular situation. The scrolling takes place at the mouse cursor's current position. Passing a positive integer scrolls up, and passing a negative integer scrolls down. Run the following in the Mu Editor's interactive shell while the mouse cursor is over the Mu Editor window:

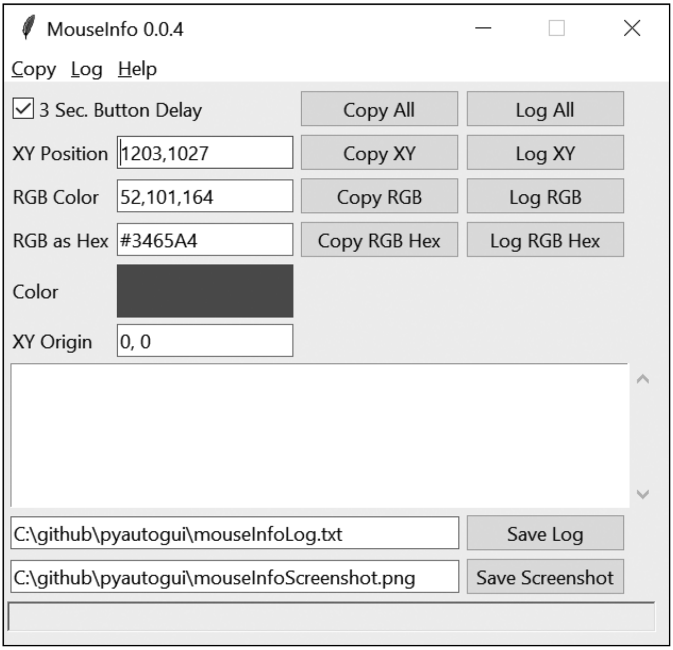
```
>>> pyautogui.scroll(200)
```

You'll see Mu scroll upward if the mouse cursor is over a text field that can be scrolled up.

Planning Your Mouse Movements

One of the difficulties of writing a program that will automate clicking the screen is finding the x- and y-coordinates of the things you'd like to click. The `pyautogui.mouseInfo()` function can help you with this.

The `pyautogui.mouseInfo()` function is meant to be called from the interactive shell, rather than as part of your program. It launches a small application named MouseInfo that's included with PyAutoGUI. The window for the application looks like Figure 23-3.



MouseInfo 0.0.4

☑ 3 Sec. Button Delay

XY Position: 1203,1027

RGB Color: 52,101,164

RGB as Hex: #3465A4

XY Origin: 0, 0

C:\github\pyautogui\mouseInfoLog.txt

C:\github\pyautogui\mouseInfoScreenshot.png

Figure 23-3: The MouseInfo application's window

Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.mouseInfo()
```

This makes the MouseInfo window appear. This window gives you information about the mouse cursor's current position, as well the color of the pixel underneath the mouse cursor, as a three-integer RGB tuple and as a hex value. The color itself appears in the color box in the window.

To help you record this coordinate or pixel information, you can click one of the eight Copy or Log buttons. The Copy All, Copy XY, Copy RGB, and Copy RGB Hex buttons will copy their respective information to the clipboard. The Log All, Log XY, Log RGB, and Log RGB Hex buttons will write their respective information to the large text field in the window. You can save the text in this log text field by clicking the Save Log button.

By default, the 3 Sec. Button Delay checkbox is checked, causing a three-second delay between clicking a Copy or Log button and the copying or logging taking place. This gives you a short amount of time in which to click the button and then move the mouse into your desired position. It may be easier to uncheck this box, move the mouse into position, and press the F1 to F8 keys to copy or log the mouse position. You can look at the Copy and Log menus at the top of the MouseInfo window to find out which key maps to which buttons.

For example, uncheck **3 Sec. Button Delay**, then move the mouse around the screen while pressing F6, and notice how the x- and y-coordinates of the mouse are recorded in the large text field in the middle of the window. You can later use these coordinates in your PyAutoGUI scripts.

For more information on MouseInfo, review the complete documentation at *https://mouseinfo.readthedocs.io/*.

Taking Screenshots

Your GUI automation programs don't have to click and type blindly. PyAutoGUI has screenshot features that can create an image file based on the current contents of the screen. These functions can also return a Pillow `Image` object of the current screen's appearance. If you've been skipping around in this book, you'll want to read Chapter 21 and install the Pillow package before continuing with this section.

To take screenshots in Python, call the `pyautogui.screenshot()` function. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> im = pyautogui.screenshot()
```

The `im` variable will contain the `Image` object of the screenshot. You can now call methods on the `Image` object in the `im` variable, just like any other `Image` object. Chapter 21 has more information about `Image` objects.

Say that one of the steps in your GUI automation program is to click a gray button. Before calling the click() method, you might want to take a screenshot and look at the pixel where the script is about to click. If it's not the same gray as the gray button, then your program knows something is wrong. Maybe the window moved unexpectedly, or maybe a pop-up dialog has blocked the button. At this point, instead of continuing, and possibly wreaking havoc by clicking the wrong thing, your program can stop itself.

You can obtain the RGB color value of a particular pixel on the screen with the pixel() function. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.pixel(0, 0)
(176, 176, 175)
>>> pyautogui.pixel((50, 200))
(130, 135, 144)
```

Pass pixel() two integers for an XY coordinate and it will tell you the color of the pixel at those coordinates in your image. The return value from pixel() is an RGB tuple of three integers for the amount of red, green, and blue in the pixel. (There is no fourth value for alpha, because screenshot images are fully opaque.)

PyAutoGUI's pixelMatchesColor() function will return True if the pixel at the given x- and y-coordinates on the screen matches the given color. The first and second arguments are integers for the x- and y-coordinates, and the third argument is a tuple of three integers for the RGB color the screen pixel must match. Enter the following into the interactive shell:

```
  >>> import pyautogui
❶ >>> pyautogui.pixel((50, 200))
  (130, 135, 144)
❷ >>> pyautogui.pixelMatchesColor(50, 200, (130, 135, 144))
  True
❸ >>> pyautogui.pixelMatchesColor(50, 200, (255, 135, 144))
  False
```

After using pixel() to get an RGB tuple for the color of a pixel at specific coordinates ❶, pass the same coordinates and RGB tuple to pixelMatchesColor() ❷, which should return True. Then, change a value in the RGB tuple and call pixelMatchesColor() again for the same coordinates ❸. This should return false. This method can be useful to call whenever your GUI automation programs are about to call click(). Note that the color at the given coordinates must match *exactly*. If it is even slightly different—for example, (255, 255, 254) instead of (255, 255, 255)—then pixelMatchesColor() will return False.

Image Recognition

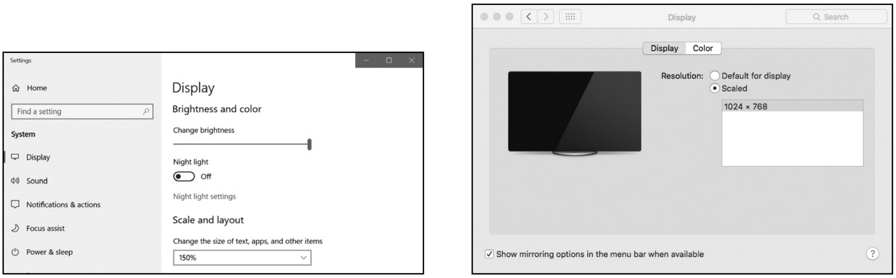
But what if you do not know beforehand where PyAutoGUI should click? You can use image recognition instead. Give PyAutoGUI an image of what you want to click, and let it figure out the coordinates.

For example, if you have previously taken a screenshot to capture the image of a Submit button in *submit.png*, the `locateOnScreen()` function will return the coordinates where that image is found. To see how `locateOnScreen()` works, try taking a screenshot of a small area on your screen; then save the image and enter the following into the interactive shell, replacing `'submit.png'` with the filename of your screenshot:

```
>>> import pyautogui
>>> box = pyautogui.locateOnScreen('submit.png')
>>> box
Box(left=643, top=745, width=70, height=29)
>>> box[0]
643
>>> box.left
643
```

The `Box` object is a named tuple that `locateOnScreen()` returns and has the x-coordinate of the left edge, the y-coordinate of the top edge, the width, and the height for the first place on the screen the image was found. If you're trying this on your computer with your own screenshot, your return value will be different from the one shown here.

If the image cannot be found on the screen, `locateOnScreen()` raises an `ImageNotFoundException`. Note that the image on the screen must match the provided image perfectly in order to be recognized. If the image is even a pixel off, `locateOnScreen()` raises an `ImageNotFoundException` exception. If you've changed your screen resolution, images from previous screenshots might not match the images on your current screen because they have a different scaling factor. Scaling factors are beyond the scope of this book, but they are sometimes used in modern, high-resolution displays. You can change the scaling in the display settings of your operating system, as shown in Figure 23-4.



Scale and layout
Change the size of text, apps, and other items
150%

Resolution: ○ Default for display ● Scaled
1024 × 768

Figure 23-4: The scale display settings in Windows (left) and macOS (right)

If the image can be found in several places on the screen, `locateAllOnScreen()` will return a `Generator` object. Generators are beyond the scope of this book, but you can pass them to `list()` to return a list of `Box` objects. There will be one `Box` object for each location where the image is found on the screen. Continue the interactive shell example by entering the following (and replacing `'submit.png'` with your own image filename):

```
>>> list(pyautogui.locateAllOnScreen('submit.png'))
[(643, 745, 70, 29), (1007, 801, 70, 29)]
```

In this example, the image appears in two locations. If your image is found in only one area, calling `list(locateAllOnScreen())` returns a list containing just one `Box` object.

Once you have the `Box` object for the specific image you want to select, you can click the center of this area by passing the tuple to `click()`. Enter the following into the interactive shell:

```
>>> pyautogui.click((643, 745, 70, 29))
```

As a shortcut, you can also pass the image filename directly to the `click()` function:

```
>>> pyautogui.click('submit.png')
```

The `moveTo()` and `dragTo()` functions also accept image filename arguments. Remember that `locateOnScreen()` raises an exception if it can't find the image on the screen, so you should call it from inside a `try` statement:

```
try:
    location = pyautogui.locateOnScreen('submit.png')
except pyautogui.ImageNotFoundException:
    print('Image could not be found.')
```

Without the `try` and `except` statements, the uncaught exception would crash your program. Since you can't be sure that your program will always find the image, it's a good idea to use the `try` and `except` statements when calling `locateOnScreen()`. In versions of PyAutoGUI before 1.0.0, `locateOnScreen()` would return `None` instead of raising an exception. Call `pyautogui.useImageNotFoundException()` in these old versions to raise an exception instead, or call `pyautogui.useImageNotFoundException(False)` for newer versions to return `None`.

Getting Window Information

Image recognition is a fragile way to find things on the screen; if a single pixel is a different color, then `pyautogui.locateOnScreen()` won't find the image. If you need to find where a particular window is on the screen, it's faster and more reliable to use PyAutoGUI's window features.

NOTE *As of version 1.0.0, PyAutoGUI's window features work only on Windows, not on macOS or Linux. These features come from PyAutoGUI's inclusion of the PyGetWindow package.*

Obtaining the Active Window

The active window on your screen is the window currently in the foreground and accepting keyboard input. If you're presently writing code in the Mu Editor, the Mu Editor's window is the active window. Of all the windows on your screen, only one will be active at a time.

In the interactive shell, call the `pyautogui.getActiveWindow()` function to get a `Window` object (technically a `Win32Window` object when run on Windows). Once you have that `Window` object, you can retrieve any of the object's attributes, which describe its size, position, and title:

`left, right, top, bottom` A single integer for the x- or y-coordinate of the window's side

`topleft, topright, bottomleft, bottomright` A `Point` named tuple of two integers for the (x, y) coordinate of the window's corner

`midleft, midright, midtop, midbottom` A `Point` named tuple of two integers for the (x, y) coordinate of the middle of the window's sides

`width, height` A single integer for one of the window's dimensions, in pixels

`size` A `Size` named tuple of two integers for the (width, height) of the window

`area` A single integer representing the area of the window, in pixels

`center` A `Point` named tuple of two integers for the (x, y) coordinate of the window's center

`centerx, centery` A single integer for the x- or y-coordinate of the window's center

`box` A `Box` named tuple of four integers for the (left, top, width, height) measurements of the window

`title` A string of the text in the title bar at the top of the window

To get the window's position, size, and title information from the `window` object, for example, enter the following into the interactive shell:

```
>>> import pyautogui
>>> active_win = pyautogui.getActiveWindow()
>>> active_win
Win32Window(hWnd=2034368)
>>> str(active_win)
'<Win32Window left="500", top="300", width="2070", height="1208", title="Mu 1.0.1 - test1.py">'
>>> active_win.title
'Mu 1.0.1 - test1.py'
>>> active_win.size
Size(width=2070, height=1208)
>>> active_win.left, active_win.top, active_win.right, active_win.bottom
```

```
(500, 300, 2570, 1508)
>>> active_win.topleft
Point(x=500, y=300)
>>> pyautogui.click(active_win.left + 10, active_win.top + 20)
```

You can now use these attributes to calculate precise coordinates within a window. If you know that a button you want to click is always 10 pixels to the right of and 20 pixels down from the window's top-left corner, and the window's top-left corner is at screen coordinates (300, 500), then calling `pyautogui.click(310, 520)` (or `pyautogui.click(active_win.left + 10, active_win.top + 20)` if `active_win` contains the `Window` object for the window) will click the button. This way, you won't have to rely on the slower, less reliable `locateOnScreen()` function to find the button for you.

Finding Windows in Other Ways

While `getActiveWindow()` is useful for obtaining the window that is active at the time of the function call, you'll need to use some other function to obtain `Window` objects for the other windows on the screen. The following three functions return a list of `Window` objects. If they're unable to find any windows, they return an empty list:

`pyautogui.getAllWindows()` Returns a list of `Window` objects for every visible window on the screen

`pyautogui.getWindowsAt(x, y)` Returns a list of `Window` objects for every visible window that includes the point (x, y)

`pyautogui.getWindowsWithTitle(title)` Returns a list of `Window` objects for every visible window that includes the string `title` in its title bar

PyAutoGUI also has a `pyautogui.getAllTitles()` function, which returns a list of strings of every visible window.

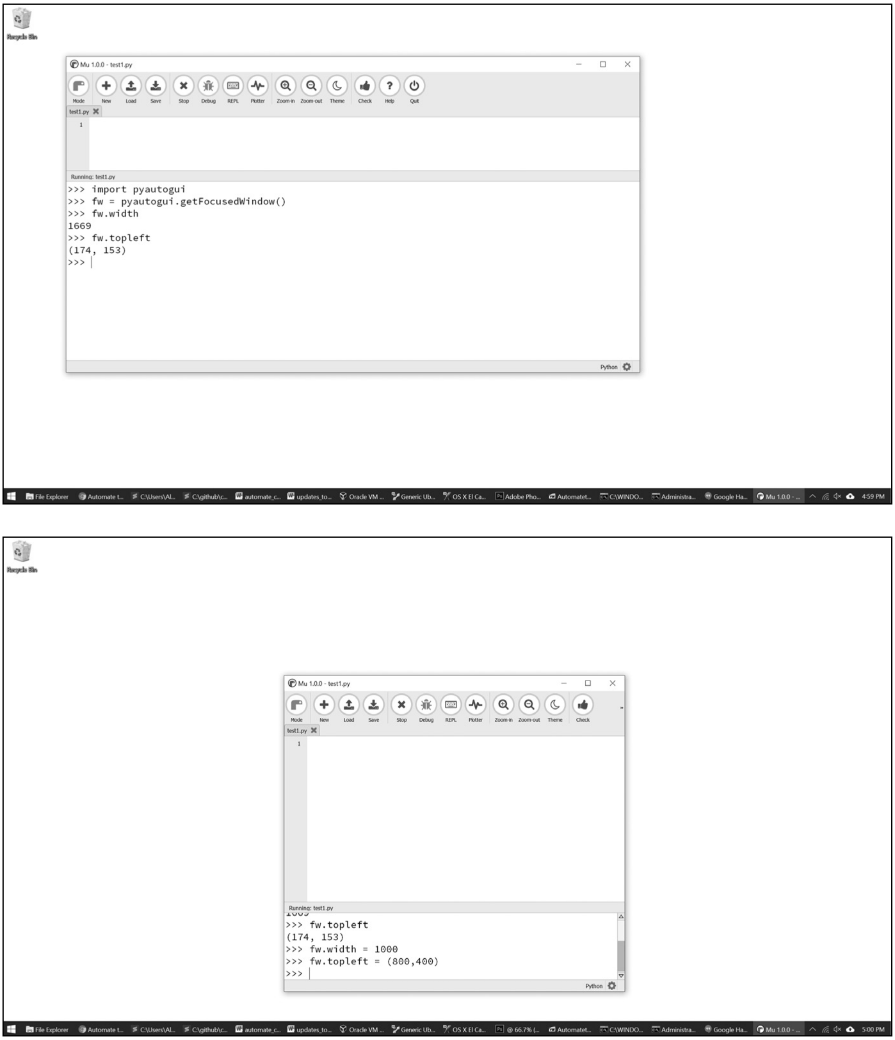
Manipulating Windows

Windows attributes can do more than just tell you the size and position of the window. You can also set their values in order to resize or move the window. For example, enter the following into the interactive shell:

```
  >>> import pyautogui
  >>> active_win = pyautogui.getActiveWindow()
❶ >>> active_win.width  # Gets the current width of the window.
  1669
❷ >>> active_win.topleft  # Gets the current position of the window.
  Point(x=174, y=153)
❸ >>> active_win.width = 1000  # Resizes the width.
❹ >>> active_win.topleft = (800, 400)  # Moves the window.
```

First, we use the `Window` object's attributes to find out information about the window's size ❶ and position ❷. After calling these functions in the

Mu Editor, the window should become narrower ❸ and move ❹, as in Figure 23-5.



```
>>> import pyautogui
>>> fw = pyautogui.getFocusedWindow()
>>> fw.width
1669
>>> fw.topleft
(174, 153)
>>>
```

```
>>> fw.topleft
(174, 153)
>>> fw.width = 1000
>>> fw.topleft = (800,400)
>>>
```

Figure 23-5: The Mu Editor window before (top) and after (bottom) using the `Window` *object attributes to resize and move it*

You can also find out and change the window's minimized, maximized, and activated states. Try entering the following into the interactive shell:

```
>>> import pyautogui
>>> active_win = pyautogui.getActiveWindow()
>>> active_win.isMaximized  # Returns True if the window is maximized
False
>>> active_win.isMinimized  # Returns True if the window is minimized
False
```

```
>>> active_win.isActive  # Returns True if the window is the active window
True
>>> active_win.maximize()  # Maximizes the window
>>> active_win.isMaximized
True
>>> active_win.restore()  # Undoes a minimize/maximize action
>>> active_win.minimize()  # Minimizes the window
>>> import time
>>> # Waits 5 seconds while you activate a different window:
>>> time.sleep(5); active_win.activate()
>>> active_win.close()  # This will close the window you're typing in.
```

The isMaximized, isMinimized, and isActive attributes contain Boolean values that indicate whether the window is currently in that state. The maximize(), minimize(), activate(), and restore() methods change the window's state. After you maximize or minimize the window with maximize() or minimize(), the restore() method will restore the window to its former size and position.

The close() method will close a window. Be careful with this method, as it may bypass any message dialogs asking you to save your work before quitting the application.

The complete documentation for PyAutoGUI's window-controlling feature can be found in its documentation.

CAPTCHAS AND COMPUTER ETHICS

Completely Automated Public Turing test to tell Computers and Humans Apart, or *captchas*, are those small tests that ask you to type the letters in a distorted picture or click photos of fire hydrants. These tests are easy, if annoying, for humans to pass but nearly impossible for software to solve. After reading this chapter, you can see how easy it is to write a script that could, say, sign up for billions of free email accounts or flood users with harassing messages. Captchas mitigate this by requiring a step that only a human can pass.

Not all websites implement captchas, however, and these sites can be vulnerable to abuse by unethical programmers. Learning to code is a powerful and exciting skill, and you may be tempted to misuse this power for personal gain or even just to show off. But just as an unlocked door isn't justification for trespass, the responsibility for your programs falls upon you, the programmer. There is nothing clever about circumventing systems to cause harm, invade privacy, or gain unfair advantage. I hope that my efforts in writing this book enable you to become your most productive self, rather than a mercenary one.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Controlling the Keyboard

PyAutoGUI also has functions for sending virtual key presses to your computer, which enables you to fill out forms or enter text into applications.

Sending Key Press Strings

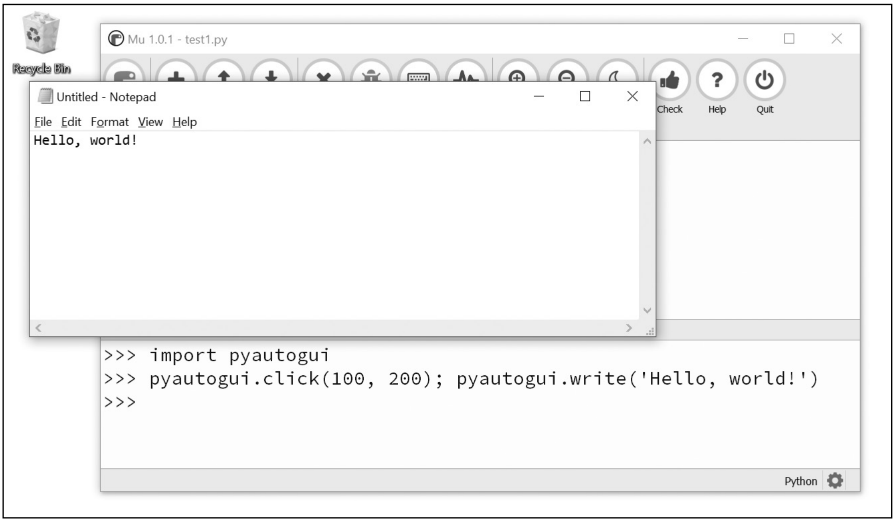
The `pyautogui.write()` function sends virtual key presses to the computer. What these key presses do depends on what window is active and what text field has focus. You may want to first send a mouse click to the text field you want in order to ensure that it has focus.

As a simple example, let's use Python to automatically type the words *Hello, world!* into a file editor window. First, open a new file editor window and position it in the upper-left corner of your screen so that PyAutoGUI will click in the right place to bring it into focus. Next, enter the following into the interactive shell:

```
>>> pyautogui.click(100, 200); pyautogui.write('Hello, world!')
```

Notice how placing two commands on the same line, separated by a semicolon, keeps the interactive shell from prompting you for input between running the two instructions. This prevents you from accidentally bringing a new window into focus between the `click()` and `write()` calls, which would mess up the example.

Python will first send a virtual mouse click to the coordinates (100, 200), which should click the file editor window and put it in focus. The `write()` call will send the text *Hello, world!* to the window, making it look like Figure 23-6. You now have code that can type for you!



```
Hello, world!
```

```
>>> import pyautogui
>>> pyautogui.click(100, 200); pyautogui.write('Hello, world!')
>>>
```

Figure 23-6: Using PyAutoGUI to click the file editor window and enter Hello, world! *into it*

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

By default, the `write()` function will enter the full string instantly. However, you can pass an optional second argument to add a short pause between each character. This second argument is an integer or float value of the number of seconds to pause. For example, `pyautogui.write('Hello, world!', 0.25)` will wait a quarter-second after typing *H*, another quarter-second after *e*, and so on. This gradual typewriter effect may be useful for slower applications that can't process keystrokes fast enough to keep up with PyAutoGUI.

For characters such as *A* or *!*, PyAutoGUI will automatically simulate holding down the SHIFT key as well.

Specifying Key Names

Not all keys are easy to represent with single text characters. For example, how do you represent SHIFT or the left arrow key as a single character? In PyAutoGUI, these keyboard keys are represented by short string values instead: `'esc'` for the ESC key or `'enter'` for the ENTER key.

Instead of a single string argument, a list of these keyboard key strings can be passed to `write()`. For example, the following call presses the A key, then the B key, then the left arrow key twice (moving the cursor in front of the "a"), and finally the X and Y keys:

```
>>> pyautogui.write(['a', 'b', 'left', 'left', 'X', 'Y'])
```

Because pressing the left arrow key moves the keyboard cursor, this will output *XYab*. Table 23-1 lists the PyAutoGUI keyboard key strings that you can pass to `write()` to simulate pressing any combination of keys.

You can also examine the `pyautogui.KEYBOARD_KEYS` list to see all possible keyboard key strings that PyAutoGUI will accept. The `'shift'` string refers to the left SHIFT key and is equivalent to `'shiftleft'`. The same applies for `'ctrl'`, `'alt'`, and `'win'` strings; they all refer to the left-side key.

Table 23-1: String Values for Keyboard Keys

Keyboard key string	Meaning
`'a'`, `'b'`, `'c'`, `'A'`, `'B'`, `'C'`, `'1'`, `'2'`, `'3'`, `'!'`, `'@'`, `'#'`, and so on	The keys for single characters
`'enter'` (or `'return'` or `'\n'`)	The ENTER key
`'esc'`	The ESC key
`'shiftleft'`, `'shiftright'`	The left and right SHIFT keys
`'altleft'`, `'altright'`	The left and right ALT keys
`'ctrlleft'`, `'ctrlright'`	The left and right CTRL keys
`'tab'` (or `'\t'`)	The TAB key
`'backspace'`, `'delete'`	The BACKSPACE and DELETE keys
`'pageup'`, `'pagedown'`	The PAGE UP and PAGE DOWN keys

(continued)

Table 23-1: String Values for Keyboard Keys *(continued)*

Keyboard key string	Meaning
`'home'`, `'end'`	The HOME and END keys
`'up'`, `'down'`, `'left'`, `'right'`	The up, down, left, and right arrow keys
`'f1'`, `'f2'`, `'f3'`, and so on	The F1 to F12 keys
`'volumemute'`, `'volumedown'`, `'volumeup'`	The mute, volume down, and volume up keys (some keyboards do not have these keys, but your operating system will still be able to understand these simulated key presses)
`'pause'`	The PAUSE key
`'capslock'`, `'numlock'`, `'scrolllock'`	The CAPS LOCK, NUM LOCK, and SCROLL LOCK keys
`'insert'`	The INS or INSERT key
`'printscreen'`	The PRTSC or PRINT SCREEN key
`'winleft'`, `'winright'`	The left and right WIN keys (on Windows)
`'command'`	The COMMAND (⌘) key (on macOS)
`'option'`	The OPTION key (on macOS)

Pressing and Releasing the Keyboard

Much like the `mouseDown()` and `mouseUp()` functions, `pyautogui.keyDown()` and `pyautogui.keyUp()` will send virtual key presses and releases to the computer. They are passed a keyboard key string (see Table 23-1) for their argument. For convenience, PyAutoGUI provides the `pyautogui.press()` function, which calls both of these functions to simulate a complete key press.

Run the following code, which will type a dollar sign (`$`) character (obtained by holding the SHIFT key and pressing 4):

```
>>> pyautogui.keyDown('shift'); pyautogui.press('4'); pyautogui.keyUp('shift')
```

This line holds down SHIFT, presses (and releases) 4, and then releases SHIFT. If you need to type a string into a text field, the `write()` function is more suitable. But for applications that take single-key commands, the `press()` function is the simpler approach.

Running Hotkey Combinations

A *hotkey* or *shortcut* is a combination of key presses to invoke some application function. The common hotkey for copying a selection is CTRL-C (on Windows and Linux) or ⌘-C (on macOS). The user presses and holds the CTRL key, then presses the C key, and then releases the C and CTRL keys.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

To do this with PyAutoGUI's `keyDown()` and `keyUp()` functions, you would have to enter the following:

```
pyautogui.keyDown('ctrl')
pyautogui.keyDown('c')
pyautogui.keyUp('c')
pyautogui.keyUp('ctrl')
```

This is rather complicated. Instead, use the `pyautogui.hotkey()` function, which takes multiple keyboard key string arguments, presses them in order, and releases them in the reverse order. For the CTRL-C example, the code would simply be as follows:

```
pyautogui.hotkey('ctrl', 'c')
```

This function is especially useful for larger hotkey combinations. In Word, the CTRL-ALT-SHIFT-S hotkey combination displays the Style pane. Instead of making eight different function calls (four `keyDown()` calls and four `keyUp()` calls), you can just call `hotkey('ctrl', 'alt', 'shift', 's')`.

Setting Up GUI Automation Scripts

GUI automation scripts are a great way to automate the boring stuff, but your scripts can also be finicky. If a window is in the wrong place on a desktop or some pop-up appears unexpectedly, your script could be clicking the wrong things on the screen. Here are some tips for setting up your GUI automation scripts:

- Use the same screen resolution each time you run the script so that the position of windows doesn't change.
- The application window that your script clicks should be maximized so that its buttons and menus are in the same place each time you run the script.
- Add generous pauses while waiting for content to load; you don't want your script to begin clicking before the application is ready.
- Use `locateOnScreen()` to find buttons and menus to click, rather than relying on coordinates. If your script can't find the thing it needs to click, stop the program rather than letting it continue blindly clicking.
- Use `getWindowsWithTitle()` to ensure that the application window you think your script is clicking exists, and use the `activate()` method to put that window in the foreground.
- Use the `logging` module from Chapter 5 to keep a logfile of what your script has done. This way, if you have to stop your script halfway through a process, you can change it to pick up from where it left off.
- Add as many checks as you can to your script. Think about how it could fail if an unexpected pop-up window appears or if your computer loses its internet connection.

- You may want to supervise the script when it first begins to ensure that it's working correctly.

You might also want to put a pause at the start of your script so that the user can set up the window the script will click on. PyAutoGUI has a `sleep()` function that acts identically to `time.sleep()` (but frees you from having to add `import time` to your scripts). There is also a `countdown()` function that prints numbers counting down to give the user a visual indication that the script will continue soon. Enter the following into the interactive shell:

```
>>> import pyautogui
>>> pyautogui.sleep(3)  # Pauses the program for 3 seconds
>>> pyautogui.countdown(10)  # Counts down over 10 seconds
10 9 8 7 6 5 4 3 2 1
>>> print('Starting in ', end=''); pyautogui.countdown(3)
Starting in 3 2 1
```

These tips can help make your GUI automation scripts easier to use and better able to recover from unforeseen circumstances.

A REVIEW OF THE PYAUTOGUI FUNCTIONS

Since this chapter covered many different functions, here is a quick summary to use as reference:

moveTo(*x, y*) Moves the mouse cursor to the given x- and y-coordinates

move(*xOffset, yOffset*) Moves the mouse cursor relative to its current position

dragTo(*x, y*) Moves the mouse cursor while the left button is held down

drag(*xOffset, yOffset*) Moves the mouse cursor relative to its current position while the left button is held down

click(*x, y, button*) Simulates a click (left button by default)

rightClick() Simulates a right-button click

middleClick() Simulates a middle-button click

doubleClick() Simulates a double left-button click

mouseDown(*x, y, button*) Simulates pressing the given button at the position *x*, *y*

mouseUp(*x, y, button*) Simulates releasing the given button at the position *x*, *y*

scroll(*units*) Simulates the scroll wheel; a positive argument scrolls up, and a negative argument scrolls down

write(*message*) Types the characters in the given message string

write([*key1, key2, key3*]) Types the given keyboard key strings

press(*key*) Presses the given keyboard key string

keyDown(*key*) Simulates pressing the given keyboard key

keyUp(*key*) Simulates releasing the given keyboard key

hotkey(*key1, key2, key3*) Simulates pressing the given keyboard key strings in order and then releasing them in reverse order

screenshot() Returns a screenshot as an `Image` object (see Chapter 21 for information on `Image` objects)

getActiveWindow(), getAllWindows(), getWindowsAt(), and getWindowsWithTitle() Return `Window` objects that can resize and reposition application windows on the desktop

getAllTitles() Returns a list of strings of the title bar text of every window on the desktop

Displaying Message Boxes

The programs you've been writing so far all tend to use plaintext output (with the `print()` function) and input (with the `input()` function). However, PyAutoGUI programs will use your entire desktop as its playground. The text-based window that your program runs in, whether it's Mu or a terminal window, will probably be lost as your PyAutoGUI program clicks and interacts with other windows. This can make getting input and output from the user difficult if the Mu or terminal window gets hidden under other windows.

To solve this, PyAutoGUI includes the PyMsgBox module to create pop-up notifications to the user and receive input from them. There are four message box functions:

pyautogui.alert(text) Displays text and has a single OK button

pyautogui.confirm(text) Displays text and has OK and Cancel buttons, returning either `'OK'` or `'Cancel'` depending on the button clicked

pyautogui.prompt(text) Displays text and has a text field for the user to type in, which it returns as a string

pyautogui.password(text) Is the same as `prompt()`, but displays asterisks so that the user can enter sensitive information such as a password

These functions are identical to the four covered in the "Pop-Up Message Boxes with PyMsgBox" in Chapter 12.

Summary

GUI automation with the PyAutoGUI package allows you to interact with applications on your computer by controlling the mouse and keyboard. While this approach is flexible enough to do anything that a human user can do, the downside is that these programs are fairly blind to what they are clicking or typing. When writing GUI automation programs, try to ensure that they will crash quickly if they're given bad instructions. Crashing is annoying, but it's much better than the program continuing in error.

You can move the mouse cursor around the screen and simulate mouse clicks, keystrokes, and keyboard shortcuts with PyAutoGUI. The PyAutoGUI package can also check the colors on the screen, which can provide your GUI automation program with enough of an idea of the screen contents to know whether it has gotten off track. You can even give PyAutoGUI a screenshot and let it figure out the coordinates of the area you want to click.

You can combine all of these PyAutoGUI features to automate any mindlessly repetitive task on your computer. In fact, it can be downright hypnotic to watch the mouse cursor move on its own and to see text appear on the screen automatically. Why not spend the time you saved by sitting back and watching your program do all your work for you? There's a certain satisfaction that comes from seeing how your cleverness has saved you from the boring stuff.

Practice Questions

1. How can you trigger PyAutoGUI's fail-safe to stop a program?
2. What function returns the current screen resolution?
3. What function returns the coordinates for the mouse cursor's current position?
4. What is the difference between `pyautogui.moveTo()` and `pyautogui.move()`?
5. What functions can be used to drag the mouse?
6. What function call will type out the characters of `"Hello, world!"`?
7. How can you do keypresses for special keys, such as the keyboard's left arrow key?
8. How can you save the current contents of the screen to an image file named *screenshot.png*?
9. What code would set a two-second pause after every PyAutoGUI function call?
10. If you want to automate clicks and keystrokes inside a web browser, should you use PyAutoGUI or Selenium?
11. What makes PyAutoGUI error-prone?
12. How can you find the size of every window on the screen that includes the word *Notepad* in its title?
13. How can you make, say, the Firefox browser active and in front of every other window on the screen?

Practice Programs

For practice, write programs to do the following tasks.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Looking Busy

Many instant messaging programs determine whether you are idle, or away from your computer, by detecting a lack of mouse movement over some period of time—say, 10 minutes. Maybe you're away from your computer but don't want others to see your instant messenger status go into idle mode to give the impression that you're slacking. Write a script to nudge your mouse cursor by one pixel to the left every 10 seconds, and then one pixel to the right 10 seconds after that. The nudge should be small and infrequent enough so that it won't get in the way if you do happen to need to use your computer while the script is running.

Reading Text Fields with the Clipboard

While you can send keystrokes to an application's text fields with `pyautogui.write()`, you can't use PyAutoGUI alone to read the text already inside a text field. This is where the `pyperclip` module can help. You can use PyAutoGUI to obtain the window for a text editor such as Mu or Notepad, bring it to the front of the screen by clicking it, click inside the text field, and then send the CTRL-A or ⌘-A hotkey to "select all" and CTRL-C or ⌘-C hotkey to "copy to clipboard." Your Python script can then read the clipboard text by running `import pyperclip` and `pyperclip.paste()`.

Write a program that follows this procedure for copying the text from a window's text fields. Use `pyautogui.getWindowsWithTitle('Notepad')` (or whichever text editor you choose) to obtain a `Window` object. The `top` and `left` attributes of this `Window` object can tell you where this window is, while the `activate()` method will ensure that it is at the front of the screen. You can then click the main text field of the text editor by adding, say, `100` or `200` pixels to the `top` and `left` attribute values with `pyautogui.click()` to put the keyboard focus there. Call `pyautogui.hotkey('ctrl', 'a')` and `pyautogui.hotkey('ctrl', 'c')` to select all the text and copy it to the clipboard. Finally, call `pyperclip.paste()` to retrieve the text from the clipboard and paste it into your Python program. From there, you can use this string however you want, but just pass it to `print()` for now.

Note that the window functions of PyAutoGUI only work on Windows as of PyAutoGUI version 1.0.0, and not on macOS or Linux.

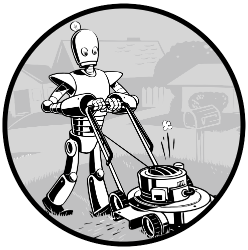
Writing a Game-Playing Bot

There is an old Flash game called Sushi Go Round. The game involves clicking the correct ingredient buttons to fill customers' sushi orders. The faster you fill orders without mistakes, the more points you get. This is a perfectly suited task for a GUI automation program—and a way to cheat to a high score! Although Flash is discontinued as a product, there are instructions for playing it offline on your computer and a list of websites that host the Sushi Go Round game at *https://github.com/asweigart/sushigoroundbot*. That GitHub repo also has the Python source code for a game-playing bot. A video of the bot playing the game is at *https://youtu.be/lfk_T6VKhTE*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

24

TEXT-TO-SPEECH AND SPEECH RECOGNITION ENGINES



This chapter covers a text-to-speech package, pyttsx3, and a speech recognition package, Whisper. Text-to-speech packages can convert text strings into spoken words, then send them to your computer's speakers or save them to an audio file. By adding this new dimension to your programs, you can free the user from having to read text off a screen. For example, a cooking recipe application could read the ingredients list aloud as you move through the kitchen, and your daily script could scrape news articles (or your emails) and then prepare an MP3 to play during your morning commute.

On the other end, speech recognition technologies can convert audio files of spoken words into text string values. You can use this capability to add voice commands to your program or automate the transcription of podcasts. And, unlike with humans, you can always mute the volume on a computer that talks too much.

Both pyttsx3 and Whisper are free to use and don't require an internet connection. The text-to-speech and speech recognition engines featured in this chapter aren't limited to English, and work with most widely spoken human languages.

Text-to-Speech Engine

To produce spoken audio, the pyttsx3 third-party package uses your operating system's built-in text-to-speech engine: Microsoft Speech API (SAPI5) on Windows, NSSpeechSynthesizer on macOS, and eSpeak on Linux. On Linux, you may need to install the engine by running `sudo apt install espeak` from a terminal window. You can install pyttsx3 by running `pip install pyttsx3` from a terminal. Appendix A has full instructions for installing third-party packages.

The name of the package is based on *py* for Python, *tts* for text-to-speech, *x* because it's extended from the original `pytts` package, and *3* because it's for Python 3.

Generating Speech

Producing a computerized voice is a complex topic in computer science. Fortunately, the operating system's text-to-speech engine does the hard work for us, and interacting with this engine is straightforward. Open a new file editor, enter the following code, and save it as *hello_tts.py*:

```
import pyttsx3
engine = pyttsx3.init()
engine.say('Hello. How are you doing?')
engine.runAndWait()  # The computer speaks.
feeling = input('>')
engine.say('Yes. I am feeling ' + feeling + ' as well.')
engine.runAndWait()  # The computer speaks again.
```

After importing the `pyttsx3` module, we call the `pyttsx3.init()` function to initialize the speech engine. This function returns an `Engine` object. We can pass a string of text to its `say()` method to tell the engine what to speak, but the actual speaking won't begin until we call the `runAndWait()` method. This method blocks (that is, will not return) until the computer has finished speaking the entire string.

The program doesn't produce any text output, because it never calls the `print()` function. Instead, you should hear your computer speak, "Hello. How are you doing?" (Make sure the volume isn't muted.) The user can enter a response from the keyboard, to which the computer should verbally reply, "Yes. I am feeling *<your response>* as well."

The pyttsx3 module allows you to make some changes to the computer voice. You can pass the strings 'rate', 'volume', and 'voices' to the getProperty() method of the Engine object to view its current settings. Enter the following into the interactive shell:

```
>>> import pyttsx3
>>> engine = pyttsx3.init()
>>> engine.getProperty('volume')
1.0
>>> engine.getProperty('rate')
200
>>> engine.getProperty('voices')
[<pyttsx3.voice.Voice object at 0x0000029DA7FB4B10>,
<pyttsx3.voice.Voice object at 0x0000029DAA3DAAD0>]
```

Note that the output may differ on your computer. The volume setting is a float, where 1.0 indicates 100 percent. The computer voice speaks at a rate of 200 words per minute. Continue this example with the following code:

```
>>> for voice in engine.getProperty('voices'):  # List all the available voices.
...     print(voice.name, voice.gender, voice.age, voice.languages)
...
Microsoft David Desktop - English (United States) None None []
Microsoft Zira Desktop - English (United States) None None []
```

On my Windows laptop with the *English (United States)* language, getProperty('voices') returns two Voice objects. (Note that this string is the plural 'voices' and not the singular 'voice'.) These Voice objects have name, gender, and age attributes, though gender and age are set to None when the operating system doesn't store that information. The languages attribute is a list of strings of languages the voice supports, which is a blank list if that information is unknown.

Let's continue the interactive shell example by calling the setProperty() method to change these settings:

```
>>> engine.setProperty('rate', 300)
>>> engine.setProperty('volume', 0.5)
>>> voices = engine.getProperty('voices')
>>> engine.setProperty('voice', voices[1].id)
>>> engine.say('The quick brown fox jumps over the yellow lazy dog.')
>>> engine.runAndWait()
```

In this example, we've changed the speaking rate to 300 words per minute and set the volume to 50 percent by passing 0.5 for the 'volume' rate. We then changed the voice to the female "Zira" voice that Windows provides by passing the id attribute of the Voice object at index 1 of the list that getProperty('voices') returned. Also note that to set the voice, we use the singular 'voice' string and not the plural 'voices' string.

Saving Speech to WAV Files

The pyttsx3 module's save_to_file() method can save the generated speech to a WAV file (with the *.wav* file extension). Enter the following into the interactive shell:

```
>>> import pyttsx3
>>> engine = pyttsx3.init()
>>> engine.save_to_file('Hello. How are you doing?', 'hello.wav')
>>> engine.runAndWait()  # The computer creates hello.wav.
```

The first argument to save_to_file() is a string of the speech to generate, while the second string argument is the filename of the *.wav* file. The text string could be a short sentence, as in the interactive shell example, or it could be pages of text. On my computer, pyttsx3 was able to turn a string of 1,800 words into a 10-minute-long audio file in about two seconds. It's important to note that calling save_to_file() alone isn't enough. You must also call the runAndWait() method before Python will create the *.wav* file.

The pyttsx3 module can save *.wav* files only, not *.mp3* files or any other audio format.

Speech Recognition

Whisper is a speech recognition system that can recognize multiple languages. Given an audio or video file, Whisper can return the speech as text in a Python string. It also returns the start and end times for groups of words, which you can use to generate subtitle files.

Install Whisper by running **pip install openai-whisper** from the terminal. (Note that the name of the speech recognition package is openai-whisper; the whisper package on the PyPI website refers to something else.) This is a large download and may take several minutes to install. Also, the first time you call the load_model() method, your computer will download the speech recognition model, which can be hundreds of megabytes or more in size.

Let's say you have an audio file named *hello.wav* in the current working directory. (Whisper can also handle *.mp3* and several other audio formats.) You could enter the following into the interactive shell:

```
>>> import whisper
>>> model = whisper.load_model('base')
>>> result = model.transcribe('hello.wav')
>>> print(result['text'])
 Hello. How are you doing?
```

After importing the whisper module, you must load the speech recognition model to use by calling the whisper.load_model() function, passing it the string of the trained machine learning model you want to use: 'tiny', 'base', 'small', 'medium', or 'large-v2'. (New models will continue to be released as well.) The smaller of these models can transcribe audio more quickly, but the larger models will do so more accurately, even when the audio has ambient noise in the background.

The first time you load a model, your computer must be connected to the internet so that the whisper module can download it from OpenAI's servers. Table 24-1 lists the model names as strings you could pass to the whisper.load_model() function, along with their file size, memory usage, and the results of some runtime tests on my laptop.

Table 24-1: Properties of Whisper Speech Recognition Models

Model name	Model file size	Estimated required memory	Runtime for a 10-word, three-second audio sample	Runtime for an 1,800-word, 15-minute audio sample
'tiny'	74MB	1GB	1.9s	1m 20s
'base'	142MB	1GB	3.0s	2m 34s
'small'	472MB	2GB	9.6s	6m 37s
'medium'	1.5GB	5GB	28.6s	20m 17s
'large-v2'	3GB	10GB	51.9s	32m 58s

My personal opinion is that for 99 percent of purposes, the 'base' model should be suitable and the 'medium' model is good enough when more accuracy is needed. All models produce errors, so the output should always undergo human review. You can experiment with larger models if you find many transcription errors in the text, or smaller models if Whisper is taking too long to transcribe the audio. As you can see in Table 24-1, however, there is a substantial difference in the two minutes and 34 seconds that the 'base' model takes to transcribe 15 minutes of audio and the nearly 33 minutes that the 'large-v2' model takes.

With the model.Whisper object that whisper.load_model() returns, you can call the transcribe() method to perform the actual transcription. Pass the method a string of the audio file's name. This method will take anywhere from a few seconds to a few hours to run, depending on the model and length of the audio file. Whisper can accept any audio or video file, and converts it to the format it requires automatically.

Whisper can automatically detect the language of the audio, but you can specify the language by passing a language keyword argument to transcribe, such as model.transcribe('hello.wav', language='English'). To find the languages that Whisper supports, you can run whisper --help from the terminal. Whisper is pretty good (but never perfect) at guessing where it should insert punctuation and at capitalizing proper names. However, you should always review the output to clean up any mistakes.

The dictionary that model.transcribe() returns has several key-value pairs, but the 'text' key contains the string of the transcription.

By default, Whisper uses your CPU to transcribe text, but if your computer has a 3D graphics card, you can greatly speed up transcriptions by setting it up to use the graphics processing unit (GPU). You'll find these setup instructions in the online documentation at *https://github.com/openai/whisper*. If your computer has an NVIDIA graphics card, you can follow the instructions in Appendix A to install packages for faster speech recognition. To use the GPU, replace the `whisper.load_model('base')` code in this chapter with `whisper.load_model('base', device='cuda')`.

You can find several additional options for Whisper in its online documentation.

Creating Subtitle Files

In addition to the transcribed audio, Whisper's results dictionary contains timing information that identifies the text's location in the audio file. You can use this text and timing data to generate subtitle files that other software can ingest. The two most common subtitle file formats are SRT SubRip Subtitle (with the *.srt* extension) and VTT Web Video Text Tracks (with the *.vtt* file extension). SRT is an older and more widespread standard, while modern video websites generally use VTT. The formats are similar. For example, here is the first part of an SRT file:

```
1
00:00:00,000 --> 00:00:05,640
Dinosaurs are a diverse group of reptiles of the clade dinosauria. They first

2
00:00:05,640 --> 00:00:14,960
appeared during the triassic period. Between 245 and 233.23 million years ago.
--snip--
```

Compare it with the first part of a VTT file for the same subtitles:

```
WEBVTT

00:00.000 --> 00:05.640
Dinosaurs are a diverse group of reptiles of the clade dinosauria. They first

00:05.640 --> 00:14.960
appeared during the triassic period. Between 245 and 233.23 million years ago.
--snip--
```

These files both indicate that the words "Dinosaurs are a diverse group . . ." appear between the start of the transcribed audio file (at 0 seconds) and the 5.640-second mark.

Whisper can also output its results as TSV data (with the *.tsv* extension) or JSON data (with the *.json* extension). TSV isn't an official subtitles format, but it may be useful if you need to export the text and timing data to, say, another Python program that can read it using the `csv` module covered in Chapter 18. TSV-formatted subtitles look like the following:

```
start   end    text
0       5640   Dinosaurs are a diverse group of reptiles of the clade dinosauria. They
5640    14960  appeared during the triassic period. Between 245 and 233.23 million years ago.
--snip--
```

To create these subtitle files, add two extra lines of code after calling `model.transcribe()`:

```
  >>> import whisper
  >>> model = whisper.load_model('base')
  >>> result = model.transcribe('hello.wav')
❶ >>> write_function = whisper.utils.get_writer('srt', '.')
❷ >>> write_function(result, 'audio')
```

The `whisper.utils.get_writer()` function ❶ accepts the subtitle file format as a string (`'srt'`, `'vtt'`, `'txt'`, `'tsv'`, or `'json'`) and the folder in which to save the file (with the `'.'` string meaning the current working directory). The `get_writer()` function returns a function to pass the transcription results. (This is a rather odd way to create the transcript files, but it's the way the `whisper` module is designed.) We store it in a variable named `write_function`, which we can then treat as a function and call, passing the `result` dictionary and the filename for the subtitle file ❷. These two lines of code produce an SRT-formatted file named *audio.srt* in the current working directory, using the text and timing information in the `result` dictionary.

Downloading Videos from Websites

While downloading audio files to transcribe with Whisper's speech recognition is often straightforward, video websites such as YouTube often don't make it easy to download their content. The `yt-dlp` module allows Python scripts to download videos from YouTube and hundreds of other video websites so that you can watch them offline. Appendix A has instructions for installing `yt-dlp`. Once it's installed, the following code will download the video at the given URL:

```
>>> import yt_dlp
>>> video_url = 'https://www.youtube.com/watch?v=kSrnLbioN6w'
>>> with yt_dlp.YoutubeDL() as ydl:
...     ydl.download([video_url])
...
```

Note that the `ydl.download()` function expects a list of video URLs, which is why we put the `video_url` string inside a list before passing it to the function call. The video's filename is based on the title on the video website, and could have a *.mp4*, *.mkv*, or other video format file extension. You'll see a lot of debugging output as the video downloads.

The video website could refuse the download due to age or login requirements, geographic restrictions, or anti-web scraping measures. If you're encountering errors, the first step you should try is installing the latest version of yt-dlp, which updates to stay compatible as video websites change their layout.

You can read about yt-dlp's many configuration options in the online documentation at *https://pypi.org/project/yt-dlp*. For example, you can extract the audio from a YouTube video by passing a dictionary of configuration settings to the yt_dlp.YoutubeDL() function:

```
>>> import yt_dlp
>>> video_url = 'https://www.youtube.com/watch?v=kSrnLbioN6w'
>>> options = {
...   ❶ 'quiet': True,  # Suppress the output.
...     'no_warnings': True,  # Suppress warnings.
...   ❷ 'outtmpl': 'downloaded_content.%(ext)s',
...     'format': 'm4a/bestaudio/best',
...     'postprocessors': [{  # Extract audio using ffmpeg.
...         'key': 'FFmpegExtractAudio',
...         'preferredcodec': 'm4a',
...     }]
... }
...
>>> with yt_dlp.YoutubeDL(options) as ydl:
...     ydl.download([video_url])
...
```

The 'quiet': True and 'no_warnings': True key-value pairs ❶ prevent the verbose debugging output. The options dictionary passed to yt_dlp.Youtube() tells it to download the video and then extract the audio to a file named *downloaded_content.m4a* ❷. (The file extension may differ if the video has a different audio format, though the *.m4a* format is the most popular one.) If we hadn't set the 'outtmpl': 'downloaded_content.%(ext)s' key-value pair in the options dictionary, the downloaded filename would be based on the video's title (excluding characters not allowed in filenames, such as question marks and colons).

To get the exact filename, we can use glob patterns, discussed in Chapter 10. We know the main part of the file is 'downloaded_content', but the file extension could be any audio format. The following code uses Path objects to find the exact downloaded filename:

```
>>> from pathlib import Path
>>> matching_filenames = list(Path().glob('downloaded_content.*'))
>>> downloaded_filename = str(matching_filenames[0])
>>> downloaded_filename
'downloaded_content.m4a'
```

Setting the filename makes it easier for the code to use this file later, such as by running it through Whisper speech recognition. The 'base' and 'medium' models create much higher-quality subtitles than the YouTube's current auto-generated subtitles.

If you just want to download the information about a given video, you can tell `yt-dlp` to skip the file and download only its metadata with the following code:

```
>>> import yt_dlp, json
>>> video_url = 'https://www.youtube.com/watch?v=kSrnLbioN6w'
>>> options = {
...     'quiet': True,  # Suppress the output.
...     'no_warnings': True,  # Suppress warnings.
...   ❶ 'skip_download': True,  # Do not download the video.
... }
...
>>> with yt_dlp.YoutubeDL(options) as ydl:
...  ❷ info = ydl.extract_info(video_url)
...  ❸ json_info = ydl.sanitize_info(info)
...     print('TITLE:', json_info['title'])  # Print the video title.
...     print('KEYS:', json_info.keys())
...     with open('metadata.json', 'w', encoding='utf-8') as json_file:
...       ❹ json_file.write(json.dumps(json_info))
...
TITLE: Beyond the Basic Stuff with Python - Al Sweigart - Part 1
KEYS: dict_keys(['id', 'title', 'formats', 'thumbnails', 'thumbnail',
'description', 'channel_id', 'channel_url', 'duration', 'view_count',
'average_rating', 'age_limit', 'webpage_url',
--snip--
```

If we just want the metadata for the video and not the video itself, we can include the `'skip_download': True` key-value pair ❶ in the `options` dictionary passed to `yt_dlp.YoutubeDL()`. The `ydl.extract_info()` method call returns a dictionary of information about the video ❷. Some of this data might not be properly formatted as JSON (discussed in Chapter 18), but we can get it in a JSON-compatible form by calling `ydl.sanitize()` ❸. The dictionary that the `sanitize()` method returns has several keys, including `'title'` for the name of the video, `'duration'` for the video's length in seconds, and so on. Our code here additionally writes this JSON data to a file named *metadata.json* ❹.

Summary

One of Python's great strengths is its vast ecosystem of third-party packages for tasks such as text-to-speech and speech recognition. These packages take some of the hardest problems in computer science and make them available to your programs with just a few lines of code.

The pyttsx3 package does text-to-speech using your computer's speech engine to create audio that you can either play from the speakers or save to a *.wav* file. The Whisper speech recognition system uses several underlying models to transcribe the words of an audio file. These models have different sizes; the smaller models transcribe faster with less accuracy, while larger models are slower but more accurate. They work for many human languages, not just English. Whisper runs on your computer and doesn't connect to online servers except to download the model on first use.

The speech engines that these Python packages use have seen a large leap in quality that wasn't available before the 2020s. Python is an excellent "glue" language that allows your scripts to connect with this software so that you can add these speech features to your own programs with just a few lines of code. If you want to learn more about text-to-speech and speech recognition, you can find many fun example projects in *Make Python Talk* by Mark Liu (No Starch Press, 2021).

Practice Questions

1. How can you make pyttsx3's voice speak faster?
2. What audio format does `pyttsx3` save to?
3. Do pyttsx3 and Whisper rely on online services?
4. Do pyttsx3 and Whisper support other languages besides English?
5. What is the name of Whisper's default machine learning model for speech recognition?
6. What are two common subtitle text file formats?
7. Can `yt-dlp` download videos from websites besides YouTube?

Practice Programs

For practice, write programs to do the following tasks.

Adding Voice to Guess the Number

Revisit the guess the number game from Chapter 3 and add a voice feature to it. Replace all of the function calls to `print()` with calls to a function named `speak()`. Next, define the `speak()` function to accept a string argument (just like `print()` did), but have it both print the string to the screen and say it out loud. For example, you'll replace this line of code

```
print('I am thinking of a number between 1 and 20.')
```

with this line of code:

```
speak('I am thinking of a number between 1 and 20.')
```

To make full use of the speech-generation feature, let's change the `'Your guess is too low.'` and `'Your guess is too high.'` text to say the player's guess. For example, the computer should say, "Your guess, 42, is too low." You can also add a voice feature to other projects in this book, such as the rock, paper, scissors game.

Singing "99 Bottles of Beer"

Cumulative songs are songs whose verses repeat with additions or slight changes. The songs "99 Bottles of Beer" and "The 12 Days of Christmas" are examples of cumulative songs. Write a program that sings (or at least speaks) the lyrics in "99 Bottles of Beer":

```
99 bottles of beer on the wall,
99 bottles of beer,
Take one down, pass it around,
98 bottles of beer on the wall.
```

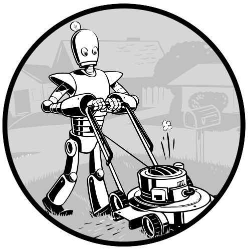
These lyrics repeat, with one fewer bottle each time. The song continues until it reaches zero bottles, at which point the last line is "No more bottles of beer on the wall." (You may wish to have the program start at 2 or 3 instead of 99 to make testing easier.)

YouTube Transcriber

Write a program that glues together the features of `yt-dlp` and Whisper to automatically download YouTube videos and produce subtitle files in the *.srt* format. The input can be a list of URLs to download and transcribe. You can also add options to produce different subtitle formats. Python is an excellent "glue language" for combining the capabilities of different modules.

A

INSTALLING THIRD-PARTY PACKAGES



Many developers have written their own modules, extending Python's capabilities beyond those provided by the standard library packaged with Python. The primary way to access third-party modules is to use Python's pip tool, which securely downloads and installs modules onto your computer from the Python Package Index (PyPI) website, a sort of free app store for Python modules. This appendix provides instructions for installing the packages used throughout the book.

For general information about working with the command line, virtual environments, packages, and modules, see Chapter 12.

Installing Pip

The executable file for the pip tool is named pip on Windows and pip3 on macOS and Linux. The pip tool has been included with Python since version 3.4. However, some distributions of Linux may not have it preinstalled.

To install pip3 on Ubuntu or Debian Linux, open a new terminal window and enter **sudo apt-get install python3-pip**. To install pip3 on Fedora Linux, enter **sudo yum install python3-pip**. These commands ask you to enter the administrator password for your computer.

Finding Pip

If pip's folder isn't included in the PATH environment variable, you may have to change directories in the terminal window with the cd command before running pip. First, find your username by running echo %USERNAME% on Windows or whoami on macOS and Linux.

Then, on Windows, run the following command, specifying your username and adjusting the folder name for the version of Python you have installed:

```
C:\Users\al>cd C:\Users\your_username\AppData\Local\Programs\Python\Python313\Scripts
```

On macOS, run this command instead (making sure to specify the correct folder name):

```
al@Als-MacBook-Pro ~ %cd /Library/Frameworks/Python.framework/Versions/3.13/bin/
```

On Linux, run this command, specifying your username:

```
al@al-VirtualBox:~$ cd /home/your_username/.local/bin/
```

You should now be in the right folder to run the pip tool. Alternatively, you can add these folders to the PATH environment variable by following the instructions in "The PATH Environment Variable" on page XX in Chapter 12.

Running pip from Virtual Environments

Some operating systems may not allow you to run pip without first creating and activating a virtual environment using Python's built-in venv module. For learning Python and experimenting with code, it's fine to create one virtual environment for all of your programs. See "Virtual Environments" on page XX in Chapter 12 for more details.

Installing the Packages Used in This Book

Because future changes to third-party modules may be incompatible with the book's code examples, I recommend that you install the exact versions used in this book by adding `==version` to the end of the module name. (Note the two equal signs in this command line option.) For example, `pip install send2trash==1.8.3` installs version 1.8.3 of the send2trash package.

The easiest way to install all compatible packages at once is to install the automateboringstuff3 package. I'll update this package as new versions of the third-party packages become available, so long as they're compatible with the book's code examples. If you'd like to install the newest versions, consult their online documentation for updated usage information.

On Windows, run the following command:

```
C:\Users\al>python -m pip install automateboringstuff3
```

On macOS and Linux, run this command:

```
al@Als-MacBook-Pro ~ % python3 -m pip install automateboringstuff3
```

To install Chapter 23's PyAutoGUI package on Linux, you must take additional steps. Open a terminal window and run **`sudo apt-get install python3-tk`** and **`sudo apt-get install python3-dev`**. To get Chapter 8's `pyperclip` module working on Linux, you must run **`sudo apt-get install xclip`**. You'll need the computer's administrator password.

As of publication, these commands install the following versions of packages:

```
beautifulsoup4==4.12.3
matplotlib==3.92
openpyxl==3.1.5
pdfminer.six==20240706
pillow==10.4.0
playsound==1.3.0
playwright==1.47.0
PyPDF==5.0.1
python-docx==1.1.2
pyttsx3==2.98
PyYAML==6.0.2
requests==2.32.3
selenium==4.25.0
tomli_w==1.0.0
xmltodict==0.13.0
```

In addition, the automateboringstuff3 package always installs the latest version of the these packages:

```
bext
ezgmail
ezsheets
humre
pyautogui
pymsgbox
pyperclip
```

```
pyperclipimg
yt-dlp
```

The Playwright package requires one more step to install separate browsers the module uses. After installing the automateboringstuff3 package, run **playwright install** to install the browsers that Playwright uses.

The pytesseract package covered in Chapter 22 and the openai-whisper package covered in Chapter 24 are fairly large, and some readers may not want to install them, so I've left them out of the automateboringstuff3 package. You can install them by running the following on Windows:

```
C:\Users\al>pip install pytesseract==0.3.10
C:\Users\al>pip install openai-whisper==20231117
```

On macOS and Linux, run the following:

```
al@Als-MacBook-Pro ~ % pip3 install pytesseract==0.3.10
al@Als-MacBook-Pro ~ % pip3 install openai-whisper==20231117
```

If you're on Windows, you might see an error like this one when installing Whisper:

```
A module that was compiled using NumPy 1.x cannot be run in
NumPy 2.0.1 as it may crash
```

You can fix this issue by running `pip install "numpy<2"` to downgrade NumPy to a compatible version. Be sure to include the double quotation marks.

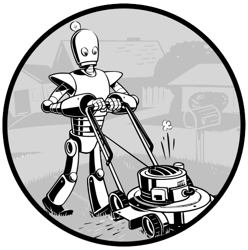
Additionally, the Whisper package can make use of your computer's NVIDIA graphics card (GPU) to perform speech recognition faster if you run this command:

```
C:\Users\al>pip install torch torchvision torchaudio --index-url
https://download.pytorch.org/whl/cu118
```

Computers with non-NVIDIA brand GPUs, including all MacBooks, don't support this feature.

B

ANSWERS TO THE PRACTICE PROBLEMS



This appendix contains the answers to the practice problems at the end of each chapter. I highly recommend that you take the time to work through these problems. Programming is more than memorizing syntax and a list of function names. As when learning a foreign language, the more practice you put into it, the more you will get out of it. There are many websites with practice programming problems as well. You can find a list of these at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*.

When it comes to the practice programs, there is no one correct solution. As long as your program performs what the project asks for, you can consider it correct. However, if you want to see examples of completed projects, they are available in the "Download the files used in the book" link at *https://nostarch.com/automate-boring-stuff-python-3rd-edition*.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Chapter 1

1. The operators are `+`, `-`, `*`, and `/`. The values are `'hello'`, `-88.8`, and `5`.
2. The string is `'spam'`; the variable is `spam`. Strings always start and end with quotes.
3. The three data types introduced in this chapter are integers, floating-point numbers, and strings.
4. An expression is a combination of values and operators. All expressions evaluate (that is, reduce) to a single value.
5. An expression evaluates to a single value. A statement does not.
6. The `bacon` variable is set to `20`. The `bacon + 1` expression does not reassign the value in `bacon` (that would need an assignment statement: `bacon = bacon + 1`).
7. Both expressions evaluate to the string `'spamspamspam'`.
8. Variable names cannot begin with a number.
9. The `int()`, `float()`, and `str()` functions will evaluate to the integer, floating-point number, and string versions of the value passed to them.
10. The expression causes an error because `99` is an integer, and only strings can be concatenated to other strings with the `+` operator. The correct way is `I have eaten ' + str(99) + ' burritos.'`

Chapter 2

1. `True` and `False`, using capital *T* and *F*, with the rest of the word in lowercase.
2. `and`, `or`, and `not`
3. `True and True` is `True`.
   `True and False` is `False`.
   `False and True` is `False`.
   `False and False` is `False`.
   `True or True` is `True`.
   `True or False` is `True`.
   `False or True` is `True`.
   `False or False` is `False`.
   `not True` is `False`.
   `not False` is `True`.
4. `False`
   `False`
   `True`
   `False`
   `False`
   `True`
5. `==`, `!=`, `<`, `>`, `<=`, and `>=`

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

6. `==` is the equal to operator that compares two values and evaluates to a Boolean, while `=` is the assignment operator that stores a value in a variable.
7. A condition is an expression used in a flow control statement that evaluates to a Boolean value.
8. The three blocks are everything inside the `if` statement and the lines `print('bacon')` and `print('ham')`:

```
print('eggs')
if spam > 5:
    print('bacon')
else:
    print('ham')
print('spam')
```

9. The code:

```
if spam == 1:
    print('Hello')
elif spam == 2:
    print('Howdy')
else:
    print('Greetings!')
```

Chapter 3

1. Press CTRL-C to stop a program stuck in an infinite loop.
2. The `break` statement will move the execution outside and just after a loop. The `continue` statement will move the execution to the start of the loop.
3. They all do the same thing. The `range(10)` call ranges from `0` up to (but not including) `10`, `range(0, 10)` explicitly tells the loop to start at `0`, and `range(0, 10, 1)` explicitly tells the loop to increase the variable by `1` on each iteration.
4. The code:

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

and:

```
i = 1
while i <= 10:
    print(i)
    i = i + 1
```

5. This function can be called with `spam.bacon()`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

Chapter 4

1. Functions reduce the need for duplicate code. This makes programs shorter, easier to read, and easier to update.
2. The code in a function executes when the function is called, not when the function is defined.
3. The `def` statement defines (that is, creates) a function.
4. A function consists of the `def` statement and the code in its `def` clause. A function call is what moves the program execution into the function, and the function call evaluates to the function's return value.
5. There is one global scope, and a local scope is created whenever a function is called.
6. When a function returns, the local scope is destroyed, and all the variables in it are forgotten.
7. A return value is the value that a function call evaluates to. Like any value, a return value can be used as part of an expression.
8. If there is no return statement for a function, its return value is `None`.
9. A `global` statement will force a variable in a function to refer to the global variable.
10. The data type of `None` is `NoneType`.
11. That `import` statement imports a module named `areallyyourpetsnamederic`. (This isn't a real Python module, by the way.)
12. This function can be called with `spam.bacon()`.
13. Place the line of code that might cause an error in a `try` clause.
14. The code that could potentially cause an error goes in the `try` clause. The code that executes if an error happens goes in the `except` clause.
15. The `random_number` global variable is set once to a random number, and the `random_number` variable in the `get_random_dice_roll()` function uses the global variable. This means the same number is returned for every `get_random_dice_roll()` function call.

Chapter 5

1. `assert spam >= 10, 'The spam variable is less than 10.'`
2. Either `assert eggs.lower() != bacon.lower() 'The eggs and bacon variables are the same!'` or `assert eggs.upper() != bacon.upper(), 'The eggs and bacon variables are the same!'`
3. `assert False, 'This assertion always triggers.'`
4. To be able to call `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
logging.basicConfig(level=logging.DEBUG, format=' %(asctime)s -
%(levelname)s - %(message)s')
```

5. To be able to send logging messages to a file named *programLog.txt* with `logging.debug()`, you must have these two lines at the start of your program:

```
import logging
logging.basicConfig(filename='programLog.txt', level=logging.DEBUG,
format=' %(asctime)s - %(levelname)s - %(message)s')
```

6. DEBUG, INFO, WARNING, ERROR, and CRITICAL
7. `logging.disable(logging.CRITICAL)`
8. You can disable logging messages without removing the logging function calls. You can selectively disable lower-level logging messages. You can create logging messages. Logging messages provides a timestamp.
9. The Step In button will move the debugger into a function call. The Step Over button will quickly execute the function call without stepping into it. The Step Out button will quickly execute the rest of the code until it steps out of the function it currently is in.
10. After you click Continue, the debugger will stop when it has reached the end of the program or a line with a breakpoint.
11. A breakpoint is a setting on a line of code that causes the debugger to pause when the program execution reaches the line.
12. To set a breakpoint in Mu, click the line number to make a red dot appear next to it.

Chapter 6

1. The empty list value, which is a list value that contains no items. This is similar to how `''` is the empty string value.
2. `spam[2] = 'hello'` (Notice that the third value in a list is at index `2` because the first index is `0`.)
3. `'d'` (Note that `'3' * 2` is the string `'33'`, which is passed to `int()` before being divided by `11`. This eventually evaluates to `3`. Expressions can be used wherever values are used.)
4. `'d'` (Negative indexes count from the end.)
5. `['a', 'b']`
6. `1`
7. `[3.14, 'cat', 11, 'cat', True, 99]`
8. `[3.14, 11, 'cat', True]`
9. The operator for list concatenation is `+`, while the operator for replication is `*`. (This is the same as for strings.)
10. While `append()` will add values only to the end of a list, `insert()` can add them anywhere in the list.
11. The `del` statement and the `remove()` list method are two ways to remove values from a list.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

12. Both lists and strings can be passed to `len()`, have indexes and slices, be used in `for` loops, be concatenated or replicated, and be used with the `in` and `not in` operators.
13. Lists are mutable; they can have values added, removed, or changed. Tuples are immutable; they cannot be changed at all. Also, tuples are written using parentheses, `(` and `)`, while lists use square brackets, [and].
14. `(42,)` (The trailing comma is mandatory.)
15. The `tuple()` and `list()` functions, respectively.
16. They contain references to list values.
17. The `copy.copy()` function will do a shallow copy of a list, while the `copy.deepcopy()` function will do a deep copy of a list. That is, only `copy.deepcopy()` will duplicate any lists inside the list.

Chapter 7

1. Two curly brackets: {}
2. `{'foo': 42}`
3. The items stored in a dictionary are unordered, while the items in a list are ordered.
4. You get a `KeyError` error.
5. There is no difference. The `in` operator checks whether a value exists as a key in the dictionary.
6. The expression `'cat' in spam` checks whether there is a `'cat'` key in the dictionary, while `'cat' in spam.values()` checks whether there is a value `'cat'` for one of the keys in `spam`.
7. `spam.setdefault('color', 'black')`
8. `pprint.pprint()`

Chapter 8

1. Escape characters represent characters in string values that would otherwise be difficult or impossible to type into code.
2. The `\n` escape character is a newline; the `\t` escape character is a tab.
3. The \\ escape character will represent a backslash character.
4. The single quote in `Howl's` is fine because you've used double quotes to mark the beginning and end of the string.
5. Multiline strings allow you to use newlines in strings without the `\n` escape character.
6. The expressions evaluate to the following:

   ```
   'e'

   'Hello'

   'Hello'

   'lo world!
   ```

7. The expressions evaluate to the following:

   `'HELLO'`

   `True`

   `'hello'`

8. The expressions evaluate to the following:

   `['Remember,', 'remember,', 'the', 'fifth', 'of', 'November.']`

   `'There-can-be-only-one.'`

9. The `rjust()`, `ljust()`, and `center()` string methods, respectively.
10. The `lstrip()` and `rstrip()` methods remove whitespace from the left and right ends of a string, respectively.

Chapter 9

1. The `re.compile()` function creates `Regex` objects.
2. Raw strings are used so that backslashes do not have to be escaped.
3. The `search()` method returns `Match` objects.
4. The `group()` method returns strings of the matched text.
5. Group `0` is the entire match, group `1` covers the first set of parentheses, and group `2` covers the second set of parentheses.
6. Periods and parentheses can be escaped with a backslash: `\.`, `\(`, and `\)`.
7. If the regex has no groups, a list of strings is returned. If the regex has groups, a list of tuples of strings is returned.
8. The `|` character signifies matching "either, or" between two groups.
9. The `?` character can either mean " match zero or one of the preceding group" or be used to signify nongreedy matching.
10. The `+` matches one or more. The `*` matches zero or more.
11. The `{3}` matches exactly three instances of the preceding group. The `{3,5}` matches between three and five instances.
12. The `\d`, `\w`, and `\s` shorthand character classes match a single digit, word, or space character, respectively.
13. The `\D`, `\W`, and `\S` shorthand character classes match a single character that is not a digit, word, or space character, respectively.
14. The `.*` performs a greedy match, and the `.*?` performs a nongreedy match.
15. Either `[0-9a-z]` or `[a-z0-9]`.
16. Passing `re.I` or `re.IGNORECASE` as the second argument to `re.compile()` will make the matching case insensitive.
17. The `.` character normally matches any character except the newline character. If `re.DOTALL` is passed as the second argument to `re.compile()`, then the dot will also match newline characters.
18. The `sub()` call will return the string `'X drummers, X pipers, five rings, X hens'`.

19. The `re.VERBOSE` argument allows you to add whitespace and comments to the string passed to `re.compile()`.

Chapter 10

1. Relative paths are relative to the current working directory.
2. Absolute paths start with the root folder, such as / or *C:\*.
3. On Windows, it evaluates to `WindowsPath('C:/Users/Al')`. On other operating systems, it evaluates to a different kind of `Path` object but with the same path.
4. The expression `'C:/Users' / 'Al'` results in an error, since you can't use the / operator to join two strings.
5. The `os.getcwd()` function *returns* the current working directory. The `os.chdir()` function *changes* the current working directory.
6. The . folder is the current folder, and .. is the parent folder.
7. *C:\\bacon\\eggs* is the directory name, while *spam.txt* is the base name.
8. The string `'r'` for read mode, `'w'` for write mode, and `'a'` for append mode.
9. An existing file opened in write mode is erased and completely overwritten.
10. The `read()` method returns the file's entire contents as a single string value. The `readlines()` method returns a list of strings, where each string is a line from the file's contents.
11. A shelf value resembles a dictionary value; it has keys and values, along with `keys()` and `values()` methods that work similarly to the dictionary methods of the same names.

Chapter 11

1. The `shutil.copy()` function will copy a single file, while `shutil.copytree()` will copy an entire folder, along with all its contents.
2. The `shutil.move()` function is used for renaming files, as well as moving them.
3. The `send2trash` function will move a file or folder to the recycle bin, while `shutil` will permanently delete files and folders.
4. The `zipfile.ZipFile()` function is equivalent to the `open()` function; the first argument is the filename, and the second argument is the mode to open the ZIP file in (read, write, or append).

Chapter 12

1. The `dir` command lists folder contents on Windows. The `ls` command lists folder contents on macOS and Linux.
2. The `PATH` environment variable contains a list of folders that are checked when a program name is entered into the terminal.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

3. The `__file__` variable contains the filename of the Python script currently being run. This variable doesn't exist in the interactive shell.
4. The `cls` command clears the terminal on Windows, while the `clear` command does so on macOS and Linux.
5. Run `python -m venv .venv` on Windows, or `python3 -m venv .venv` on macOS and Linux.
6. Run `python -m PyInstaller --onefile` *`yourScript.py`* on Windows, or `python3 -m PyInstaller --onefile` *`yourScript.py`* on macOS and Linux.

Chapter 13

1. The `webbrowser` module has an `open()` method that will launch a web browser to a specific URL, and that's it. The `requests` module can download files and pages from the web. The `bs4` module parses HTML.
2. The `requests.get()` function returns a `Response` object, which has a `text` attribute that contains the downloaded content as a string.
3. The `raise_for_status()` method raises an exception if the download had problems and does nothing if the download succeeded.
4. The `status_code` attribute of the `Response` object contains the HTTP status code.
5. After opening the new file on your computer in `'wb'` "write binary" mode, use a `for` loop that iterates over the `Response` object's `iter_content()` method to write out chunks to the file. Here's an example:

```
saveFile = open('filename.html', 'wb')
for chunk in res.iter_content(100000):
 saveFile.write(chunk)
```

6. Most online APIs return their responses formatted as JSON or XML.
7. F12 brings up the developer tools in Chrome. Pressing CTRL-SHIFT-C (on Windows and Linux) or ⌘-OPTION-C (on OS X) brings up the developer tools in Firefox.
8. Right-click the element in the page and select Inspect Element from the menu.
9. `'#main'`
10. `'.highlight'`
11. `spam.getText()`
12. `linkElem.attrs`
13. The `selenium` module is imported with `from selenium import webdriver`.
14. The `find_element_*` methods return the first matching element as a `WebElement` object. The `find_elements_*` methods return a list of all matching elements as `WebElement` objects.
15. The `click()` and `send_keys()` methods simulate mouse clicks and keyboard keys, respectively.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

16. The `press('Control+A')` method simulates pressing CTRL+A.
17. The `forward()`, `back()`, and `refresh()` `WebDriver` object methods simulate these browser buttons.
18. The `go_forward()`, `go_back()`, and `reload()Page` object methods simulate these browser buttons.

Chapter 14

1. The `openpyxl.load_workbook()` function returns a `Workbook` object.
2. The `sheetnames` attribute contains a list of strings of the worksheet titles.
3. Run `wb['Sheet1']`.
4. Use `wb.active`.
5. `sheet['C5'].value` or `sheet.cell(row=5, column=3).value`
6. `sheet['C5'] = 'Hello'` or `sheet.cell(row=5, column=3).value = 'Hello'`
7. `cell.row` and `cell.column`
8. They hold the highest column and row with values in the sheet, respectively, as integer values.
9. `openpyxl.cell.column_index_from_string('M')`
10. `openpyxl.cell.get_column_letter(14)`
11. `sheet['A1':'F1']`
12. `wb.save('example3.xlsx')`
13. A formula is set the same way as any value. Set the cell's `value` attribute to a string of the formula text. Remember that formulas begin with the equal sign (`=`).
14. Pass `data_only=True` when calling `load_workbook()` to make OpenPyXL retrieve the calculated results of formulas.
15. `sheet.row_dimensions[5].height = 100`
16. `sheet.column_dimensions['C'].hidden = True`
17. Freeze panes are rows and columns that will always appear on the screen. They are useful for headers.
18. `openpyxl.chart.Reference()`, `openpyxl.chart.Series()`, `openpyxl.chart.BarChart()`, `chartObj.append(seriesObj)`, and `add_chart()`

Chapter 15

1. To access Google Sheets, you need a credentials file, a token file for Google Sheets, and a token file for Google Drive.
2. EZSheets has `ezsheets.Spreadsheet` and `ezsheets.Sheet` objects.
3. Call the `downloadAsExcel()` `Spreadsheet` method.
4. Call the `ezsheets.upload()` function and pass the filename of the Excel file.
5. Read the value at `ss['Students']['B2']`.
6. Call `ezsheets.getColumnLetterOf(999)`.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

7. Access the `rowCount` and `columnCount` properties of the `Sheet` object.
8. Call the `delete()` `Sheet` method. This is only permanent if you pass the `permanent=True` keyword argument.
9. The `createSpreadsheet()` function and `createSheet()` `Spreadsheet` method will create `Spreadsheet` and `Sheet` objects, respectively.
10. EZSheets will throttle your method calls.

Chapter 16

1. `conn = sqlite3.connect('example.db', isolation_level=None)`
2. `conn.execute('CREATE TABLE students (first_name TEXT, last_name TEXT, favorite_color TEXT) STRICT')`
3. Pass the `isolation_level=None` keyword argument when calling `sqlite3.connect()`.
4. While `INTEGER` is analogous to Python's int type, `REAL` is analogous to Python's float type.
5. Strict mode adds a requirement that every column must have a data type, and SQLite raises an exception if you try to insert data of the wrong type.
6. The `*` means "select all columns in the table."
7. CRUD stands for Create, Read, Update, and Delete, the four standard operations that databases perform.
8. ACID stands for Atomic, Consistent, Isolated, and Durable, the four properties that database transactions should have.
9. `INSERT` queries add new records to tables.
10. `DELETE` queries delete records from tables.
11. Without a `WHERE` clause, the `UPDATE` query applies to all records in the table, which may or may not be what you want.
12. An index is a data structure that organizes a column's data, which takes up more storage but makes queries faster. `'CREATE INDEX idx_birthdate ON cats (birthdate)'` would create an index for the `cats` table's `birthdate` column.
13. A foreign key links records in one table to a record in another table.
14. You can delete a table named `cats` by running the query `'DROP TABLE cats'`.
15. The string `':memory:'` is used in place of a filename to create an in-memory database.
16. The `iterdump()` method can create the queries to copy a database. You can also copy the database file itself.

Chapter 17

1. A `File` object must be opened in write mode by passing `'w'` to `open()`.
2. Calling `getPage(4)` will return a `Page` object for page 5, since page 0 is the first page.

3. Call `decrypt('swordfish')`.
4. Rotate pages counterclockwise by passing a negative integer: `-90`, `-180`, or `-270`.
5. `docx.Document('demo.docx')`
6. Use `doc.paragraphs` to obtain a list of `Paragraph` objects.
7. A `Paragraph` object represents a paragraph of text and is itself made up of one or more `Run` objects.
8. A `Run` object has these variables (*not* a `Paragraph` object).
9. `True` always makes the `Run` object bolded and `False` always makes it not bolded, no matter what the style's bold setting is. `None` will make the `Run` object just use the style's bold setting.
10. Call the `docx.Document()` function.
11. `doc.add_paragraph('Hello there!')`
12. The integers `0` through `9`.

Chapter 18

1. In Excel, spreadsheets can have values of data types other than strings; cells can have different fonts, sizes, or color settings; cells can have varying widths and heights; adjacent cells can be merged; and you can embed images and charts.
2. You pass a `File` object, obtained from a call to `open()`.
3. `File` objects need to be opened in read-binary (`'rb'`) mode for `Reader` objects and write-binary (`'wb'`) mode for `Writer` objects.
4. The `writerow()` method.
5. The `delimiter` argument changes the string used to separate cells in a row. The `lineterminator` argument changes the string used to separate rows.
6. All of them can be easily edited with a text editor: CSV, JSON, and XML are plaintext formats.
7. `json.loads()`
8. `json.dumps()`
9. XML's format resembles HTML.
10. JSON represents `None` values as the keyword `null`.
11. Boolean values in JSON are written in lowercase: `true` and `false`.

Chapter 19

1. A reference moment that many date and time programs use. The moment is January 1, 1970, UTC.
2. `time.time()`
3. `time.asctime()`
4. `time.sleep(5)`

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

5. It returns the closest integer to the argument passed. For example, `round(2.4)` returns `2`.
6. A `datetime` object represents a specific moment in time. A `timedelta` object represents a duration of time.
7. Run `datetime.datetime(2019, 1, 7).weekday()`, which returns `0`. This means Monday, as the `datetime` module uses `0` for Monday, `1` for Tuesday, and so on, up to `6` for Sunday.

Chapter 20

1. The *credentials.json* and *token.json* files tell the EZGmail module which Google account to use when accessing Gmail.
2. A message represents a single email, while a back-and-forth conversation involving multiple emails is a thread.
3. Include the `'has:attachment'` text in the string you pass to `search()`.
4. SMS email gateways are not guaranteed to work, don't notify you if the message was delivered, and just because they worked before does not mean they will work again.
5. The Requests library can send and receive ntfy notifications.

Chapter 21

1. An RGBA value is a tuple of four integers, each ranging from 0 to 255. The four integers correspond to the amount of red, green, blue, and alpha (transparency) in the color.
2. Calling `ImageColor.getcolor('CornflowerBlue', 'RGBA')` will return `(100, 149, 237, 255)`, the RGBA value for the cornflower blue color.
3. A box tuple is a tuple value of four integers: the left-edge x-coordinate, the top-edge y-coordinate, the width, and the height, respectively.
4. `Image.open('zophie.png')`
5. `im.size` is a tuple of two integers, the width and the height.
6. `im.crop((0, 50, 50, 50))`. Notice that you are passing a box tuple to `crop()`, not four separate integer arguments.
7. Call the `im.save('new_filename.png')` method of the `Image` object.
8. The `ImageDraw` module contains code to draw on images.
9. `ImageDraw` objects have shape-drawing methods such as `point()`, `line()`, or `rectangle()`. They are returned by passing the `Image` object to the `ImageDraw.Draw()` function.
10. `plt.plot()` creates a line graph, `plt.scatter()` creates a scatterplot, `plt.bar()` creates a bar graph, and `plt.pie()` creates a pie chart.
11. The `savefig()` method saves the graph as an image.
12. You cannot call `plt.show()` twice in a row because it resets the graph data, forcing you to run the graph-making code again if you want to show it a second time.

Chapter 22

1. Tesseract recognizes English by default.
2. PyTesseract works with the Pillow image library.
3. The `image_to_string()` function accepts an `Image` object and returns a string.
4. No, Tesseract only extracts text from scanned documents of typewritten text and not text from photographs.
5. `tess.get_languages()` returns a list of language pack strings.
6. You can pass the `lang='eng+jpn'` keyword argument to identify both English and Japanese text in an image.
7. NAPS2 can be run from a Python script to create PDFs with embedded OCR text.

Chapter 23

1. Move the mouse to any corner of the screen.
2. The `pyautogui.size()` function returns a tuple with two integers for the width and height of the screen.
3. The `pyautogui.position()` function returns a tuple with two integers for the x- and y-coordinates of the mouse cursor.
4. The `moveTo()` function moves the mouse to absolute coordinates on the screen, while the `move()` function moves the mouse relative to the mouse's current position.
5. `pyautogui.dragTo()` and `pyautogui.drag()`
6. `pyautogui.typewrite('Hello world!')`
7. Either pass a list of keyboard key strings to `pyautogui.write()` (such as `'left'`) or pass a single keyboard key string to `pyautogui.press()`.
8. `pyautogui.screenshot('screenshot.png')`
9. `pyautogui.PAUSE = 2`
10. You should use Selenium for controlling a web browser instead of PyAutoGUI.
11. PyAutoGUI clicks and types blindly and cannot easily find out if it's clicking and typing into the correct windows. Unexpected pop-up windows or errors can throw the script off track and require you to shut it down.
12. Call the `pyautogui.getWindowsWithTitle('Notepad')` function.
13. Run `w = pyatuogui.getWindowsWithTitle('Firefox')`, and then run `w.activate()`.

Chapter 24

1. Call `engine.setProperty('rate', 300)`, for example, to make pyttsx3's voice speak at 300 words per minute.

Automate the Boring Stuff with Python, 3rd edition (Early Access) © 2025 by Al Sweigart

2. The `pyttsx3` module saves to the WAV audio format.
3. No, pyttsx3 and Whisper do not require an online service or internet access.
4. Yes, pyttsx3 and Whisper support languages other than English.
5. Whisper's default model is `'base'`.
6. SRT (SubRip Subtitle) and VTT Web Video Text Tracks are two common subtitle file formats.
7. Yes, `yt-dlp` can download from hundreds of video websites other than YouTube.