EBOOK EXTRAS: v3.4
Downloads, Updates, Feedback

TAKE CONTROL OF

The Mac Command Line with Terminal

by JOE KISSELL

$14.99

3RD EDITION

Table of Contents

Read Me First

Welcome to *Take Control of the Mac Command Line with Terminal, Third Edition,* version 3.4, published in January 2025 by alt concepts. This book was written by Joe Kissell and edited by Geoff Duncan.

This book introduces you to the macOS command line environment, teaching you how to use the Terminal utility to accomplish useful, interesting tasks that are either difficult or impossible to perform in the graphical interface. Most of the examples work with 10.6 Snow Leopard and later, although a few techniques require more recent versions of macOS.

If you want to share this ebook with a friend, we ask that you do so as you would with a physical book: "lend" it for a quick look, but ask your friend to buy a copy for careful reading or reference. Discounted classroom and user group copies are available.

Updates and More

You can access extras related to this ebook on the web (use the link in Ebook Extras, near the end; it's available only to purchasers). On the ebook's Take Control Extras page, you can:

- Download any available new version of the ebook for free, or buy any subsequent edition at a discount.
- Access the book in both PDF and EPUB formats. (Learn about reading on mobile devices on our Device Advice page.)
- Read the ebook's blog. You may find new tips or information, as well as a link to an author interview.

If you bought this ebook from the Take Control website, it has been added to your account, where you can download it in other formats and access any future updates.

Basics

Please be aware of the following special considerations:

- **Spurious hyphens!** When you view this ebook in EPUB format, your ebook reader (such as Apple Books or Kindle) may insert extra hyphens in the longer lines of text that are provided as examples of what to type on the command line. You can mitigate this problem by viewing the text in a single column, with a smaller font, and in a landscape orientation. In some cases, you can turn off auto-hyphenation to remove these spurious hyphens. For example, if you are reading in Books in iOS, you can go to Settings > Books and turn off the Auto-hyphenation switch. However, with auto-hyphenation off, Books may now cut off some wider lines of command-line text.

  If you are reading this ebook to absorb the material conceptually, this won't be a problem, but if you want to type the commands on your Mac, consider downloading the PDF of this ebook onto your Mac, in order to read it there. As a bonus, you can copy the command-line text out of the PDF and paste it on the command line. Read Ebook Extras for help with downloading the PDF.

- **Entering commands:** I frequently tell you to "enter" a command in a Terminal window. This means you should type the command *and then press Return or Enter*. Typing a command without pressing Return or Enter afterward has no effect.

- **Getting commands into Terminal:** When you see commands that are to be entered into a Terminal window, you can type them manually. As I mentioned just above, if you're reading this on a Mac, you can copy the command from the ebook and paste it into Terminal (which is handy, especially for longer and more complex commands).

  Whichever method you use, keep these tips in mind:

  - *When typing:* Every character counts, so watch carefully. The font that represents text you should type is *monospaced,* meaning every character has the same width. So, if it looks like there's a space between two characters, there is—and you should type it. Similarly, be sure to type all punctuation—such as hyphens, periods, and quotation marks—exactly as it appears in the book, even if it seems odd. If you type the wrong thing, the command probably won't work. (In the EPUB version of this book, the font shown might not be monospaced. Also, be sure to *read the first item in this list,* to avoid entering unnecessary hyphens.)

- *When copying and pasting:* If you select a line of text to copy and paste into Terminal, be sure that your selection begins with the first character and ends with the last. If you accidentally leave out characters, the command probably won't work, and if you select too much (for example, extending your selection to the next line), you may see unexpected results, such as the command executing before you're ready.

Settings vs. Preferences

In macOS Ventura, Apple replaced System Preferences with System Settings, and in most apps (including Terminal), what was formerly a Preferences window is now a Settings window. In this book, I generally use a shorthand like "go to Terminal > Settings/Preferences" or "open System Preferences/System Settings" to reflect these two possibilities. When I list System Settings separately, that applies only to Ventura or later.

What's New in Version 3.4

Version 3.4 makes a few minor corrections and improvements to scattered commands based on reader feedback, and adds the topic What Changed in Sequoia? (basically, nothing of consequence).

What Was New in Version 3.3

Version 3.3 was a minor update that refreshed the book slightly for macOS 14 Sonoma and current versions of third-party tools. Changes included:

- Added the topic What Changed in Sonoma? (spoiler: hardly anything)
- Made some minor adjustments throughout to account for small user interface changes, and to put information about newer Apple hardware ahead of information about older hardware
- Updated the sidebar !! Plus to cover the `$!` shortcut
- Updated a variety of details in Use a Package Manager; more specifically, in Homebrew, removed several caveats and qualifications that no longer apply, since the software was updated to behave differently
- Added a new recipe: Enable or Disable Screen Sharing Remotely

Introduction

Back when I began using computers, in the early 1980s, user interfaces were pretty primitive. A computer usually came with only a keyboard for input—mice were a novelty that hadn't caught on yet. To get your computer to do something, you typed a command, waited for some result, and then typed another command. There was simply no concept of pointing and clicking to make things happen.

When I finally switched from DOS to the Mac (without ever going through a Windows phase, I should mention!), I was thrilled that I could do my work without having to memorize lists of commands, consult manuals constantly, or guess at how to accomplish something. Everything was right there on the screen, just a click away. It was simpler—not in the sense of being less powerful, but in the sense of requiring less effort to access the same amount of power. Like most everyone else, I fell instantly in love with graphical interfaces.

Fast forward a few decades, and I sometimes find myself faced with some mundane task, such as deleting a file that refuses to disappear from the Trash or changing an obscure system preference. After wasting time puzzling over how to accomplish my task—and perhaps doing some web searches—I discover that the Mac's graphical interface does not, in fact, offer any built-in way to do what I want. So I have to hunt on the internet for an app that seems to do what I want, download it, install it, and run it (and perhaps pay for it, too), all so that I can accomplish a task with my mouse that would have taken me 5 seconds in DOS 30-odd years ago.

That's not simple.

I'm a Mac user because I don't have time to waste. I don't want my computer to put barriers between me and my work. I want easier ways to do things instead of harder ways. Ironically, the Mac's beautiful graphical interface, with all its menus, icons, and buttons, doesn't always provide the easiest way to do something, and in some cases it doesn't even provide a hard way. The cost of elegance and simplicity is sometimes a lack of flexibility.

Luckily, macOS isn't restricted to the graphical realm of windows and icons. It has another whole interface that lets you accomplish many tasks that would otherwise be difficult, or even impossible. This other way of using macOS looks strikingly like those DOS screens from the 1980s: it's a command-line interface, in which input is done with the keyboard, and the output is sent to the screen in plain text.

The usual way of getting to this alternative interface (though there are others) is to use an app called Terminal, located in the Utilities folder inside your Applications folder. It's a simple app that doesn't appear to do much at first glance—it displays a window with a little bit of text in it. But Terminal is in fact the gateway to vast power.

If you read TidBITS, Take Control books, Macworld, or any of the numerous other Mac publications, you've undoubtedly seen tips from time to time that begin, "Open Terminal and type in the following…". Many Mac users find that sort of thing intimidating. What do I *click*? How do I find my way around? How do I stop something I've started? Without the visual cues of a graphical interface, lots of people get stuck staring at that blank window.

If you're one of those people, this book is for you. It's also for people who know a little bit about the command line but don't fully understand what they can do, how to get around, and how to stay out of trouble. By the time you're finished reading this book and trying out the examples I give, you should be comfortable interacting with your Mac by way of the command line, ready to confidently use Terminal whenever the need arises.

It's not scary. It's not hard. It's just different. And don't worry—I'll be with you every step of the way!

Much of this book is concerned with teaching you the skills and basic commands you must know in order to accomplish genuinely useful things later on. If you feel that it's a bit boring or irrelevant to learn how to list files or change directories, remember: it's all about the end result. You learn the fundamentals of baking not because measuring flour or preheating an oven is intrinsically interesting, but because you need to know how to do those things in order to end up with cookies. And let me tell you, the cookies make it all worthwhile!

Speaking of food—my all-purpose metaphor—this book doesn't *only* provide information on individual ingredients and techniques. In particular, the last chapter is full of terrific, simple command-line recipes that put all this power to good use while giving you a taste of some advanced capabilities I don't explore in detail. Among many other things, this book shows you:

- How to figure out what's preventing a disk from disconnecting (unmounting or ejecting)
- How to tell which apps are currently accessing the internet
- How to rename lots of files at once, even if you're not running Yosemite or later
- How to change a number of hidden preferences
- How to understand and change file permissions
- How to automate command-line activities with scripts

Astute readers may note that some of these tasks can be accomplished with third-party utilities. That's true, but the command line is infinitely more flexible—and Terminal is free!

I should be clear, however, that this book won't turn you into a command-line expert. I would need thousands of pages to describe everything you can accomplish with the command line. Instead, my goal is to cover the basics and get you up to a moderate level of familiarity and competence. And, based on feedback from the first two editions of this book, I've expanded the scope of this revised third edition to include a number of topics that are a bit more advanced.

Most of my examples work with any version of macOS from 10.6 Snow Leopard on, although a few techniques require later versions; I point those out as we go along.

macOS Command Line Quick Start

This book is mostly linear—the later sections tend to build on the earlier sections. For that reason, I strongly recommend starting from the beginning and working through the book in order (perhaps skimming lightly over any sections that explain already familiar concepts). You can use the items in the final chapter, Command-Line Recipes, at any time, but they'll make more sense if you understand all the basics presented earlier in the book.

Find your bearings:

- Learn about the command line and its terminology; see Understand Basic Command-Line Concepts.
- Become familiar with the most common tool for accessing the command line; see Get to Know (and Customize) Terminal.
- Navigate using the command line; see Look Around.

Learn basic skills:

- Create, delete, and modify files and directories; see Work with Files and Directories.
- Run or stop programs and scripts; see Work with Programs.
- Make your command-line environment work more efficiently; see Customize Your Defaults.

Go beyond the Terminal window:

- Integrate the command line and the Mac's graphical interface; see Bring the Command Line into the Real World.
- Use the command line to control another Mac; see Log In to Another Computer.

Earn your propeller beanie:

- Learn about users, groups, permissions, and the infamous `sudo` command; see Work with Permissions.
- Learn Advanced Techniques such as piping and redirecting data, using the `grep` search tool, and adding logic to your shell scripts.
- Go beyond what's built into macOS by downloading third-party command-line programs; see Install New Software.
- Find quicker and easier ways to perform common activities in Terminal; see Learn Command-Line Shortcuts.

Put your skills into practice:

- Do cool (and practical) stuff on the command line; see Command-Line Recipes.

Understand Basic Command-Line Concepts

In order to make sense of what you read about the command line, you should know a bit of background material. This chapter explains the ideas and terminology I use throughout the book, providing context for everything I discuss later.

What's Unix?

Unix is a computer operating system with roots going back to 1969. Back then, Unix referred to one specific operating system running on certain expensive minicomputers (which weren't "mini" at all; they were enormous!). Over time, quite a few companies, educational institutions, and other groups have developed their own variants of Unix—some were offshoots from the original version and others were built from scratch.

After many branches, splits, mergers, and parallel projects, there are now more than a dozen distinct families of Unix and Unix-like operating systems. Within each family, such as Linux (a Unix-like system), there may be many individual variants, or distributions.

Note: A Unix-like system is one that looks and acts like Unix, but doesn't adhere completely to a list of standards known as the Single UNIX Specification, or SUS. Mac OS X 10.5 Leopard or later running on an Intel-based Mac is a true Unix operating system. Earlier versions of Mac OS X, and any version running on PowerPC-based Macs, were technically Unix-like.

macOS is a version of Unix that nicely illustrates this process of branching and merging. On the one hand, you had the classic Macintosh OS, which developed on its own path between 1984 and 2002. On the other hand, you had NeXTSTEP, an operating system based on a variety of Unix called BSD (Berkeley Software Distribution). NeXT, the developer of NeXTSTEP, was the company that Steve Jobs founded after leaving Apple in 1985.

When Apple bought NeXT in 1996, they began building a new operating system that extended and enhanced NeXTSTEP while layering on capabilities (and some of the user interface) of the classic Mac OS. The result was Mac OS X (later renamed OS X and now macOS): it's Unix underneath, but with lots of extra stuff that's not in other versions of Unix. If you took macOS and stripped off the graphical interface, the Cocoa application programming interfaces (APIs), and all the built-in apps such as Mail and Safari, you'd get the Unix core of macOS. This core has its own name: Darwin. When you work in the command-line environment, you'll encounter this term from time to time.

Darwin is itself a complete operating system, and though Apple doesn't sell computers that run only Darwin, it is available as open source so anyone with sufficient technical skill can download, compile, and run Darwin as an operating system on their own computer—for free.

What's a Command Line?

A command-line interface is a way of giving instructions to a computer and getting results back. You type a *command* (a word or other sequence of characters) and press Return or Enter. The computer then processes that command and displays the result (often in a list or other chunk of text). In most cases, all your input and output remains on the screen, scrolling up as more appears. But only one line—usually the last line of text in the window, and usually designated by a blinking cursor—is the actual *command line,* the one where commands appear when you type them.

Note: Although Darwin (which has only a command-line interface) is part of macOS, it isn't quite correct to say that you're working in Darwin when you're using the macOS command line. In fact, the command line gives you a way to interact with all of macOS, only part of which is Darwin.

What’s a Shell?

A *shell* is a program that creates a user interface of one kind or another, enabling you to interact with a computer. In macOS, the Finder is a type of shell—a graphical shell—and there are still other varieties with other interfaces. But for the purposes of this book, I use the term “shell” to refer only to programs that create a command-line interface.

macOS includes up to seven different shells, which means that your Mac has not just one command-line interface, but seven! These shells share many attributes—in fact, they’re more alike than different. Most commands work the same way in all the shells, and produce similar results. The shells in macOS are all standard Unix shells, and at least one of them is on pretty much any computer running any Unix or Unix-like operating system.

The original Unix shell was called the Bourne shell (after its creator, Stephen Bourne). The actual program that runs the Bourne shell has a much shorter name: `sh`. The other Unix shells included with macOS are:

- **csh:** the C shell, named for similarities to the C programming language (Unix folks love names with puns, too, as you’ll see)
- **tcsh:** the Tenex C shell, which adds features to `csh`
- **ksh:** the Korn shell, a variant of `sh` (with some `csh` features) developed by David Korn
- **bash:** the Bourne-again shell (yet another superset of `sh`)
- **zsh:** the Z shell, an advanced shell named after Yale professor Zhong Shao that incorporates features from `tcsh`, `ksh`, and `bash`, plus other capabilities
- **dash:** the Debian Almquist shell, a lightweight shell that’s been around for more than two decades, but was not included with macOS until Catalina

In Mac OS X 10.2 Jaguar and earlier versions, `tcsh` was the default shell. Starting with 10.3 Panther, `bash` became the new default. And in 10.15 Catalina, `zsh` replaced `bash` as the default shell (see Zsh Became the New Default Shell). However, if you've upgraded your Mac (or transferred your user account) from an earlier version of macOS that used a different default shell, your account may still be configured to use the earlier default shell.

In this book, I discuss only the `zsh` and `bash` shells (which, as you'll see, are similar enough that most of what I say applies equally to both of them). Almost everything you learn in this book will serve you well regardless of which shell you use—`zsh`, `bash`, or any of the others.

Tip: You don't have to stick with any of Apple's preinstalled shells. You are free to download and install others, too! One rather popular alternative shell (which I'm afraid I can't discuss in any detail in this book) is fish, for "friendly interactive shell." It's both powerful and user-friendly, and if you end up spending a lot of time on the command line, it's worth checking into.

A bit later in the book, in Set a Default Shell, I show you how to confirm which shell you're currently using and how to change your default, if you like.

What's Terminal?

So, how do you run a shell in order to use a command-line interface on your Mac? You use an app called a *terminal emulator*.

As the name suggests, a terminal emulator simulates a *terminal*—the devices people used to interact with computers back in the days of monolithic mainframes. A terminal consisted of little more than a display (or, even earlier, a printer), a keyboard, and a network connection. Terminals may have looked like computers, but all they did was receive input from users, send it along to the actual computer (which was likely in a different room or even a different building), and display any results that came back.

A modern terminal emulator program provides a terminal-like connection to a shell running either on the same computer or on a different computer over a network.

Quite a few terminal emulators run on macOS, but the one you're most likely to use is called—you guessed it—Terminal, and it's included as part of macOS. Some third-party terminal emulators, such as iTerm 2, have fancy and impressive features that will save you time and effort if you do a lot of work on the command line. (Read about even more options in 12 Best Alternative Terminal Emulators for Mac at Geeks-Mint.) However, in order to keep this book to a reasonable length, I discuss only Terminal here.

Terminal Commands? Not Really!

At the risk of redundancy, I want to emphasize where Terminal fits into the scheme of things. A common misconception is that Terminal *is* the macOS command-line interface. You'll hear people talk about entering "Terminal commands" and things of that sort. (Even I have said things like that from time to time.) But that's incorrect. Terminal is just an app—one of numerous similar apps—that gives you access to the Mac's command-line interface. When you run a command-line program, you're running it in a shell, which in turn runs in Terminal.

So, to summarize: you use Terminal to run a shell, which provides a command-line interface to macOS—a variety of Unix (of which the non-graphical portion is known as Darwin). You can use the macOS command line successfully without having all those facts entirely clear in your mind, but a rough grasp of the hierarchy makes the process a bit more comprehensible.

Tip: Although I do my best in this book to acquaint you with what I consider the most important features of Terminal, you may also want to consult Apple's official Terminal User Guide on the web for more information.

What Are Commands, Arguments, and Flags?

The last piece of background information I want to provide has to do with the kinds of things you type into a Terminal window. I provide extensive examples of all these items ahead, but I want to give you an introduction to three important terms: *commands, arguments,* and *flags*. If you don't fully understand this stuff right now, don't worry: it will become clearer after some examples.

Commands

Commands are straightforward; they're the verbs of the command line (even though they may look nothing like English verbs). When you enter a command, you tell the computer to do something, such as run a program. Very often, entering a command—a single word or abbreviation—is sufficient to get something done.

Note: As a reminder, when I say "enter this," I mean "type this, and then press Return or Enter."

For example—not to get ahead of myself but just to illustrate—if you enter the command `date`, your Terminal window shows the current date and time.

Note: Many commands are abbreviations or shortened forms of longer terms—for example, the command `pwd` stands for Print Working Directory.

Arguments

Along with commands (verbs), we have *arguments*, which you can think of as nouns—or, in grammatical terms, direct objects. For example, I could say to you, "Eat!," and you could follow that command by consuming any food at hand. However, if I want you to eat something in particular, I might say, "Eat cereal!" Here, *cereal* is the direct object, or what we'd call an argument in a command-line interface.

On the command line, you must frequently specify the file, directory, or other item to which you want a command applied. In general, you simply type the command, a space, and then the argument. For example, the command `nano`, by itself, opens a text editor called `nano`. (In other words, entering `nano` means "*run* `nano`"—you tell the shell to execute a command simply by entering its name.) But enter `nano file1` and the command instead opens the file `file1` using the `nano` text editor. Here, `file1` is the argument to the command `nano`.

> **Note:** Always be sure to type a space after the command and before any arguments.

Some commands accept no arguments. Some take optional arguments. And some commands require one or even several arguments. For example, to change the modification date of three files—`file1`, `file2`, and `file3`—I can enter `touch file1 file2 file3`. But other commands require multiple arguments that have different meanings (as in "Process `file1` with the information found in `file2` and store the output in `file3`"). In these cases, the order in which the arguments appear is critical. I detail which commands in this book take arguments, the order of those arguments, and the circumstances when you need to use those arguments.

Flags

Besides verbs and nouns, we have adverbs! In English, I could say, "Eat cereal *quickly*!" or "Watch TV *quietly*." The adverbs *quickly* and *quietly* don't tell you what to do, but rather how to do it. By analogy, an expression in a command-line statement that specifies how a command should be accomplished is called a *flag*, though you may also hear it referred to as an *option* or *switch*. (Some people consider a flag to be a type of argument, but I'm going to ignore that technicality.)

Suppose I want to list the files in a directory. I could enter the `ls` ("list") command, which would do just that. But if I want to list the files in a particular way—say, in a way that included their sizes and modification dates—I could add a flag to the `ls` command.

The flag that `ls` uses to indicate a "long" listing (including sizes and dates) is `-l`. So if I enter `ls -l` (note the space before the flag), I get the kind of listing I want.

Flagging Enthusiasm

I should mention a couple of irritations with flags:

- First, you'll notice in this example that the flag was preceded by a hyphen: `-l`. That's common, and it enables the command to distinguish a flag (which has a hyphen) from an argument (which doesn't). Unfortunately, Unix commands aren't entirely consistent. You'll sometimes see commands that require flags with no hyphen, commands that require flags with two hyphens, and commands with flags that can appear in either a "short" form (one hyphen, usually followed by a single letter) or a "long" form (two hyphens, usually followed by a complete word).
- Second, a command may take more than one flag. ("Eat quickly and quietly!") For example, you might want to tell the `ls` command not only to use the long format (`-l`) but also to show all files, including any hidden ones (`-a`, which you can think of as "all" or "anything"). Here you get two choices. You can either combine the flags (`ls -la` or `ls -al`) or keep them separate (`ls -l -a` or `ls -a -l`). In this example, both ways work just fine, and the flags work in any order. But that isn't always the case; some commands are picky and require you to list flags one way or the other.

Don't worry about these differences; just be aware that they may come up from time to time. For now, assume that most flags start with a single hyphen, and that the safest way to express most flags is to keep them separate.

Some commands require both arguments and flags. In general, the order is `command flag(s) argument(s)`, which is unlike usual English word order—it would be comparable to saying, "Eat quickly cereal!" For example, if you want to use the `ls` ("list") command to show you only the names of files beginning with the letter r (`r*`), in long (`-l`) format, you'd put it like this: `ls -l r*`.

Sin Tax?

As you read about the command line, you'll sometimes see the word *syntax*, which is a compact way of saying, "which arguments and flags are required for a given command, which are optional, and what order they should go in." When I say that the usual order is `command flag(s) argument(s)`, I'm making a general statement about syntax, though there are plenty of exceptions.

One place you see a command's syntax spelled out is in the `man` (manual) pages for Unix programs (see Get Help), at the top under the heading "Synopsis." For example, the `man` page for the `mkdir` (make directory) command (see Create a Directory) gives the following:

```
mkdir [-pv] [-m mode] directory-name ...
```

Here's how to read this command's syntax, one item at a time (don't worry about exactly what each item does; this is just for illustration):

- `mkdir`: First is the command itself.
- `[-pv]`: Anything in brackets is optional, and if possible, flags are run together in the syntax when using the command. So we know that the `-p` flag and the `-v` flag are both optional, and if you want to use them both, they can optionally be written as `-pv`.
- `[-m mode]`: Another optional flag is `-m`, and it's listed separately because if you do use it, it requires its own argument (another string of characters, described in the `man` page). The underline beneath `mode` means it's a variable; you have to fill in the mode that you want.
- `directory-name`: This argument is not optional (because it's not in brackets), and it's also a variable, meaning you supply your own value.
- `...`: Finally, we have an underlined ellipsis, which simply means you can add on more arguments like the last one. In this case, it would mean you could list additional directories to be created.

So the final command could look like, for example:

`mkdir teas` (all optional items omitted), or

`mkdir -pv -m 777 a/b/teas a/b/nuts` (all optional items included).

What Changed in Catalina?

For most of the history of macOS, changes to the command-line environment (that is, the Terminal app, the shells themselves, and the preinstalled Unix programs) have been minor and infrequent. Indeed, if you were to use the very first version of this book (from 2009) and try the commands in today's version of macOS, the vast majority of them would still work, just as most of the examples in this book work as far back as the very first release of Mac OS X.

However, the 2019 release of macOS 10.15 Catalina brought several noteworthy changes, and still more occurred in macOS 11 Big Sur (see What Changed in Big Sur?) and macOS 12 Monterey (see What Changed in Monterey?). Let's start with the Catalina changes.

Zsh Became the New Default Shell

Catalina changed the default shell, for the first time since Panther in 2003: `zsh` is now the default, rather than `bash`. This change, which was widely misunderstood, had almost zero effect on the overwhelming majority of users. Almost everyone can ignore it, almost all the time.

The facts are these:

- **Zsh has been in macOS a long time.** As I hinted earlier in What's a Shell?, both `zsh` and `bash` have been part of macOS for many years. You could have used `zsh` in Mac OS X 10.1 in 2001 if you wanted to! It's not a new shell, and it's not new to the Mac.
- **Bash is still there.** The `bash` shell is still included with Catalina (and later), and it still works just fine. Apple didn't remove it—they just changed the default—but it is possible Apple could remove it some future version of macOS.
- **Your default stays in place when you upgrade.** If `bash` was your default shell in Mojave (or earlier) and you upgraded your Mac to Catalina (or later), bash is still your default. `Zsh` is the default only for new installations of Catalina (or later), or newly created user accounts.

- **Zsh and bash are extremely similar.** Nearly everything you can do in `bash` works exactly the same way in `zsh`. Sure, each shell has its idiosyncrasies and `zsh` has some nice additional features; I discuss some of them later in Zsh Tips and Shortcuts. But unless you're a hard-core command-line user, you may never notice these changes.

- **"Default" just means what runs automatically.** When you open the Terminal app (or you open a new tab or window in Terminal), the shell you've set as the default runs. That's all "default" means.

- **You can change your default easily.** I describe multiple ways to do this later on, in Set a Default Shell.

- **You can switch shells anytime, on the fly.** Regardless of your default, you can switch shells instantly, at any time, simply by typing the other shell's name. So, if you're currently in a `bash` shell and you decide you want to use `zsh`, just enter `zsh`. If you're in a `zsh` shell and you want to use `bash`, just enter `bash`. Done!

Note: If you normally use `zsh` but occasionally switch to `bash`, you may find after entering `bash` that your `.bash_profile` customizations don't load. You can load them by entering `source .bash_profile`.

- **Scripts aren't necessarily affected at all.** If you write shell scripts (see Create Your Own Shell Script), or use shell scripts other people have written, those scripts should start with a line that says which shell should perform the script's actions. A script that specifies the `bash` shell will continue to work even if you're running it in `zsh`, and vice versa. But for that matter, script syntax between the two shells is so similar that most scripts could switch their specified shell and still work exactly the same.

- **You can follow (or ignore) prompts to switch.** If you've upgraded to Catalina or later from an earlier version of macOS that used a different default, you'll see something like this in every new Terminal window:

  ```
  The default interactive shell is now zsh.
  To update your account to use zsh, please run `chsh -s /bin/zsh`.
  For more details, please visit https://support.apple.com/kb/HT208050.
  ```

  If you want to go ahead and switch your shell to `zsh`, you can follow those instructions. But if you want to keep using your old shell—and *not* be constantly nagged about it—you can suppress that message. Read How to hide the 'default interactive shell is now zsh' message in Terminal on macOS at AddictiveTips for instructions.

- **Recovery mode still uses bash.** If you reboot your Mac into recovery mode (see About Recovery Mode) and choose Utilities > Terminal open the Terminal app there, you'll find that `bash` is still the default in that environment (and `zsh` isn't even available as an option, though `sh` and `dash` are). This is still true in Monterey.

All that said, since `zsh` and `bash` are so similar, I include instructions for both of them in this book. Except as noted, everything I say here works exactly the same way regardless of which of those two shells you use.

The Startup Volume Became Read-Only

In order to reduce the possibility that malware (or, ahem, user error) could damage, delete, or replace essential system files, Apple made a big change starting in Catalina. The startup volume is now split into two volumes behind the scenes—one (still called Macintosh HD by default) that's read-only and contains the core components of macOS, and another that's writable (called Macintosh HD - Data by default) and contains all your optional and third-party apps, user-generated data, and so on. The upshot of this arrangement is that it's next to impossible to make changes to your startup volume (intentionally or otherwise) that would prevent your Mac from booting—and that's a good thing!

Apple did some clever tricks to hide the complexity of the two-volume arrangement. In the Finder as well as on the command line, this split is normally invisible—ordinarily, you'll never see the "Data" volume separately, and the `/Applications` folder (for example) shows both Apple's apps and third-party apps as though they were all located on the same volume. However, in some situations, especially when working on the command line or writing scripts, you have to be able to distinguish which of the two underlying volumes a certain file or folder is really on, and you have to know how to make changes to the writeable "Data" volume. I discuss this situation in the context of the `ls` ("list") command—refer to See What's Here—but the instructions there can apply to any command.

Scripting Languages Were Deprecated

Scripting languages (notably Perl, PHP, Python, Ruby, and Tcl) were included with macOS from the very beginning, and they're commonly found on other Unix and Unix-like operating systems (see Run a Script). Their presence not only enables you to run and write scripts in these languages, but also makes it possible for lots of important third-party software to run. (For example, the popular Homebrew system, which I discuss in Use a Package Manager, is written entirely in Ruby, and therefore requires Ruby to be present in order to run.)

In Catalina, Apple *deprecated* these languages, which meant, basically, that although they were still present and still worked for the time being, Apple was declaring that those languages had essentially been abandoned in place and would be removed in the future (see What Changed in Monterey? for details). Developers whose software depends on them were urged to bundle the relevant scripting languages with their apps.

I don't know why Apple did this (though I have some theories), and it's unfortunate because it has caused (and continues to cause) a considerable amount of confusion and frustration. However, it's ultimately not a huge issue, for two reasons:

- **You can still download and install these languages yourself.** If you find yourself on a Mac without one of these languages (or with an older version), you can download and install the latest version yourself. See How to Download Scripting Languages, later in this chapter, for details.

- **macOS shipped with old versions anyway.** For reasons only Apple knows, the versions of these languages that shipped with macOS were always significantly behind the current release versions. For that reason, serious scripters often downloaded newer releases anyway—either supplementing or replacing the ones Apple provided.

If you're running Big Sur or earlier, you can basically ignore all this for now—unless you want to install a newer version of one of these languages for some reason. Just be aware that in Monterey or later, you may have to jump through some extra hoops to be able to use certain command-line software that depends on them.

Note: Apple says they're deprecating "scripting language runtimes such as Python, Ruby, and Perl," but that is certainly not a complete list, as PHP and Tcl are also deprecated. It's unclear where Apple intends to draw the line—AppleScript and JavaScript are scripting languages, and arguably so are most shells (including `zsh` and `bash`), but I don't expect those to disappear.

Disappearing Unix Programs

While Apple has deprecated scripting languages, other Unix programs long included with macOS were unceremoniously dumped starting in High Sierra—most prominently, `ftp` (used for file transfer) and `telnet` (used to run remote shell sessions). Both of these programs are wildly insecure, and both have much better and more modern alternatives (`sftp` and `ssh`, respectively), which still are included with macOS. However, if you should find yourself needing a plain FTP client, you can download a program called `ncftp` using any of the three package managers I discuss in Use a Package Manager, and if you need `telnet` for something `ssh` can't do, you can download it using either the Homebrew or Fink package managers.

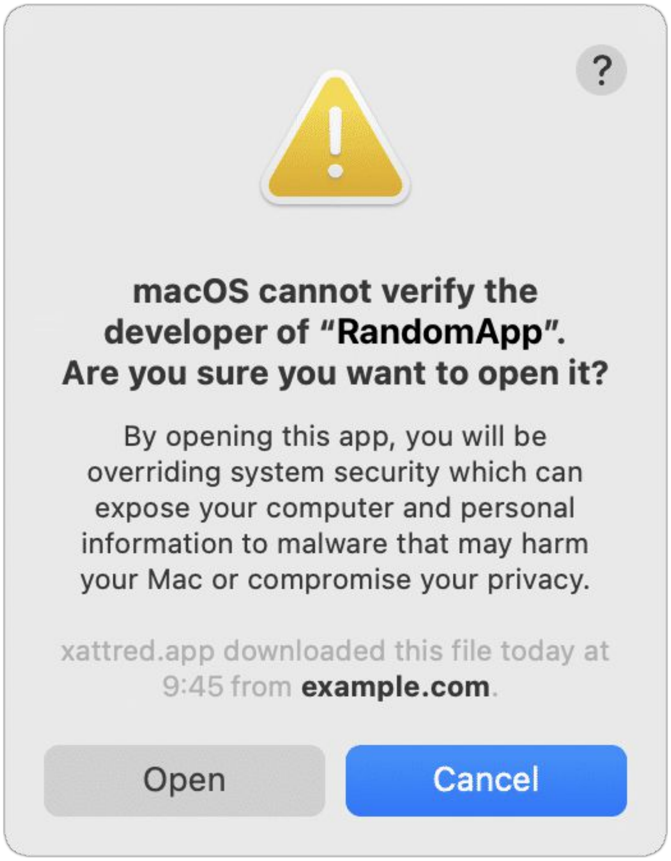
Apple Began Enforcing Notarization Requirement

In Apple's ongoing quest to reduce the likelihood of malware infecting your Mac, the company has been imposing progressively stricter requirements on developers. Now, I could spend several pages here explaining all the complexities of terms like *code-signing*, *hardening*, and *notarization*, but I'll spare you. After all, if you're an Apple developer, you undoubtedly know about all that stuff already, and if you aren't a developer, almost none of those details will have any effect on your day-to-day computer usage. Ordinary users should not have to lose sleep over whether some software they downloaded was notarized or not—or even what notarization is. (OK, since you're twisting my arm, it's basically Apple's stamp of approval that software has passed an automated malware check.)

Except…

On February 3, 2020, Apple began enforcing a requirement that they'd previously relaxed. For certain kinds of software built after June 1, 2019 and distributed on the internet (but outside the App Store), users running Catalina or later will see an error message unless the developer has gone through the extra step of notarization. That can include command-line software, though the *vast* majority of command-line software you download (see Install New Software) will either be old enough that this requirement doesn't apply or new enough that developers have notarized it. You'll *hardly ever* run into problems.

Nevertheless, it *could* occur that you download, install, and run a new command-line program only to see a warning like the one in **Figure 1**.



macOS cannot verify the developer of "RandomApp". Are you sure you want to open it?

By opening this app, you will be overriding system security which can expose your computer and personal information to malware that may harm your Mac or compromise your privacy.

xattred.app downloaded this file today at 9:45 from **example.com**.

Figure 1: A non-notarized command-line app named `piffle` would produce an error message like this one when you try to run it.

Should this happen to you, *don't worry*. As long as you are confident that you downloaded the software from a reputable source (or are willing to take the small risk of encountering malware), you can bypass this warning:

1. Click Cancel in the initial alert—*not* Move to Trash!

2. Open System Settings > Privacy & Security or System Preferences > Security & Privacy > General.

3. At the bottom, you should see a message along these lines:

   "*Program name*" was blocked from use because it is not from an identified developer.

Next to that is a button that says Open Anyway. Click it!

4. Try running the app again. This time you'll get a slightly different warning:

   macOS cannot verify the developer of "*program name*". Are you sure you want to open it?

   Click Open.

The program then runs, and macOS won't pester you about that program in the future.

What Changed in Big Sur?

macOS 11 Big Sur included all the changes mentioned above from Catalina, but little else of consequence when it comes to the command line. However, you should be aware of the following:

- **Signed system volume:** Catalina introduced the read-only system volume (see The Startup Volume Became Read-Only); Big Sur upped the ante by cryptographically signing that volume, making it even harder for malware to do any mischief. However, in terms of following the instructions in this book for using the command line, that change is effectively invisible.
- **Startup chime:** Apple brought back the startup chime in Big Sur, even for Macs that had never had it enabled. If you don't like it, though, you can turn it off—and you can enable it on Macs that aren't running Big Sur. Use the instructions in Enable or Disable Your Mac's Startup Chime.
- **Support for Apple silicon:** Apple's latest Macs feature an entirely new processor and system architecture. Amazingly, most software made for Intel-based Macs still works on these new machines, and that includes command-line software. On my own M1 MacBook Air, the only thing I had to do was update Homebrew to the latest version, which has native Apple silicon support. Although there are probably a few older command-line apps that won't work on M-

series Macs without an update, virtually nothing else covered in this book has changed, except a new way to enter recovery mode (see About Recovery Mode).

- **Scripting languages:** Surprisingly, the scripting languages deprecated in Catalina (see Scripting Languages Were Deprecated) were still present in Big Sur, but as I explain below, they did in fact start to disappear in Monterey.

What Changed in Monterey?

Of the huge number of changes in Monterey, only two are directly applicable to the material in this book.

PHP and Python Disappear

The main thing Apple changed in macOS 12 Monterey was to begin following through on their promise to remove deprecated scripting languages (see Scripting Languages Were Deprecated). The initial release of Monterey dropped PHP, while 12.3 removed Python.

As I mentioned, if you're running Monterey (or later) and you need PHP, it's no big deal—you can easily download and install it (see How to Download Scripting Languages). The same is true of Python, although the situation is a bit more muddled there.

As Ben Toms notes in macOS Monterey 12.3 Will Remove Python 2.7, the version of Python Apple had previously included with macOS was 2.7, and the entire 2.x branch was sunsetted by the Python Foundation in 2020. That means if you're going to install Python yourself, you'll want to install version 3.x. Unfortunately, due to changes in the language, not all scripts written for Python 2.x work under 3.x without modifications. Moreover, some Python software may be hard-coded to look for the Python executable in the spot where Apple used to put it. However, that location is invalid once you upgrade, and macOS's increased security features make it infeasible to add a simple redirection.

Long story short: if you were using Python software prior to Monterey, you may have to do some recoding to get that software working under a self-installed copy of Python 3.

Proxy Icons Return (Sorta)

You no longer have to do a command-line dance to display proxy icons; all you have to do now is check a box in System Settings/System Preferences. See Remove the Proxy Icon Hover Delay in Big Sur.

Special Permissions for Terminal and the Command Line

Starting in Mojave, Apple placed additional restrictions on what resources apps can access without a user's express permission, and that list is now longer. As you use the command line, you may be prompted one or more times to grant Terminal (or a command-line program running within it) access to system resources.

For example:

- Some operations that use the `sudo` command (see Perform Actions as the Root User) work only if you've added Terminal to the list of apps in System Settings > Privacy & Security > Full Disk Access or System Preferences > Security & Privacy > Privacy > Full Disk Access—*and* enabled it. (If you enter a command and see the text "Operation not permitted," that's a sure sign you need to do this.)
- In some rare situations, you may be prompted to grant extra privacy permissions for a particular *shell* (like `sh` or `zsh`).
- Starting in Catalina, if you use the `screencapture` program on the command line to take screenshots or movies of your screen, you're prompted to add Terminal to the list of approved apps in System Settings > Privacy & Security > Screen Recording or System Preferences > Security & Privacy > Privacy > Screen Recording. (This also requires quitting and reopening Terminal.)
- Some operations on the command line (or in third-party apps that support shell scripts) may require authorization in System Settings > Privacy & Security > Automation or System Preferences > Security & Privacy > Privacy > Automation.

If a shell says you're not permitted to do something, even using `sudo`, the first thing to investigate is whether all the relevant checkboxes for Terminal and command-line apps are selected in these places.

Apple Replaces nano with pico

In Monterey 12.3, Apple swapped out the `nano` text editor for `pico`, putting in a link so that you can still type `nano` to open a text editor—and that editor will look and work almost exactly like `nano`. For more on this odd change, see the sidebar From pico to nano and Back.

What Changed in Ventura?

macOS 13 Ventura changed almost nothing when it comes to the Mac command line. As of macOS 13.1, Ruby and Perl are still included (see Scripting Languages Were Deprecated), and the only change I've noticed that affects this book has to do with Preview, which lost the capability to read PostScript files, thus breaking one of my recipes. For a new, improved version of that recipe, see Read man Pages in Preview.

What Changed in Sonoma?

As far as I can tell, almost nothing pertaining to the command line has changed in macOS 14 Sonoma. Notably, given Apple's earlier moves regarding scripting languages (see Scripting Languages Were Deprecated and PHP and Python Disappear), Sonoma still contains built-in versions of Perl, Ruby, and Tcl (outdated though they may be).

One minor change is that it's now possible to not only change a setting that lets you use Touch ID in place of typing a password when you enter `sudo`, but also enable that change to persist across macOS updates; see the tip in Using sudo. There are also a few small changes geared toward developers that ordinary users will never encounter.

It's possible that some other command-line programs or behaviors were removed or changed without my noticing, but in the months that I've been using Sonoma, I haven't encountered any commands, scripts, or recipes breaking.

What Changed in Sequoia?

What I said above about Sonoma remains true in Sequoia: as of version 15.3, I have seen no changes to the command line that affect this book. (Some minor changes to tools for developers and enterprise administrators have occurred, but that's beyond the scope of what this book addresses.)

How to Download Scripting Languages

As I mentioned in Scripting Languages Were Deprecated, the removal of scripting languages from macOS—which actually started in Monterey with PHP and, later, Python—is not a crisis for several reasons, not the least of which is that anyone can download and install these languages on a Mac quickly and at no cost. And, as a bonus, you'll get a more recent version than Apple would have included anyway.

To install a scripting language, you can use either the links below or a package manager (see Use a Package Manager):

- **Perl:** Download the ActiveState Perl installer, or (on the same page) even customize an installation for your specific needs.
- **PHP:** Install PHP using the package manager of your choice. Alternatively, you can download the source code and compile it yourself.
- **Python:** Download the current release (or earlier releases) from python.org.
- **Ruby:** Download any of several different installers, or download the source code and compile it yourself, at ruby-lang.org.
- **Tcl:** Download the ActiveState Tcl installer from ActiveState.

About Recovery Mode

Recovery mode (formerly known as macOS Recovery) is a special way of running your Mac that lets you perform troubleshooting steps, restore a backup, or reinstall macOS—even if your startup volume is damaged. Terminal is available in this mode, so you can also use a number of command-line tools (covered later in this book).

If you have an M-series Mac, here's how you enter recovery mode:

1. Shut down your Mac completely.
2. Press and hold the power button until you see the message "Loading startup options…" beneath the Apple logo, then release it.
3. Wait until the screen finishes loading, which may take a minute or two. Then select the Options icon and click the Continue button below it.

To enter recovery mode on an Intel-based Mac, restart your Mac, immediately hold down ⌘-R, and release the keys when the Apple logo appears.

With either system architecture, if you have enabled FileVault, you must then select a user, click Next, enter that user's password, and click Continue.

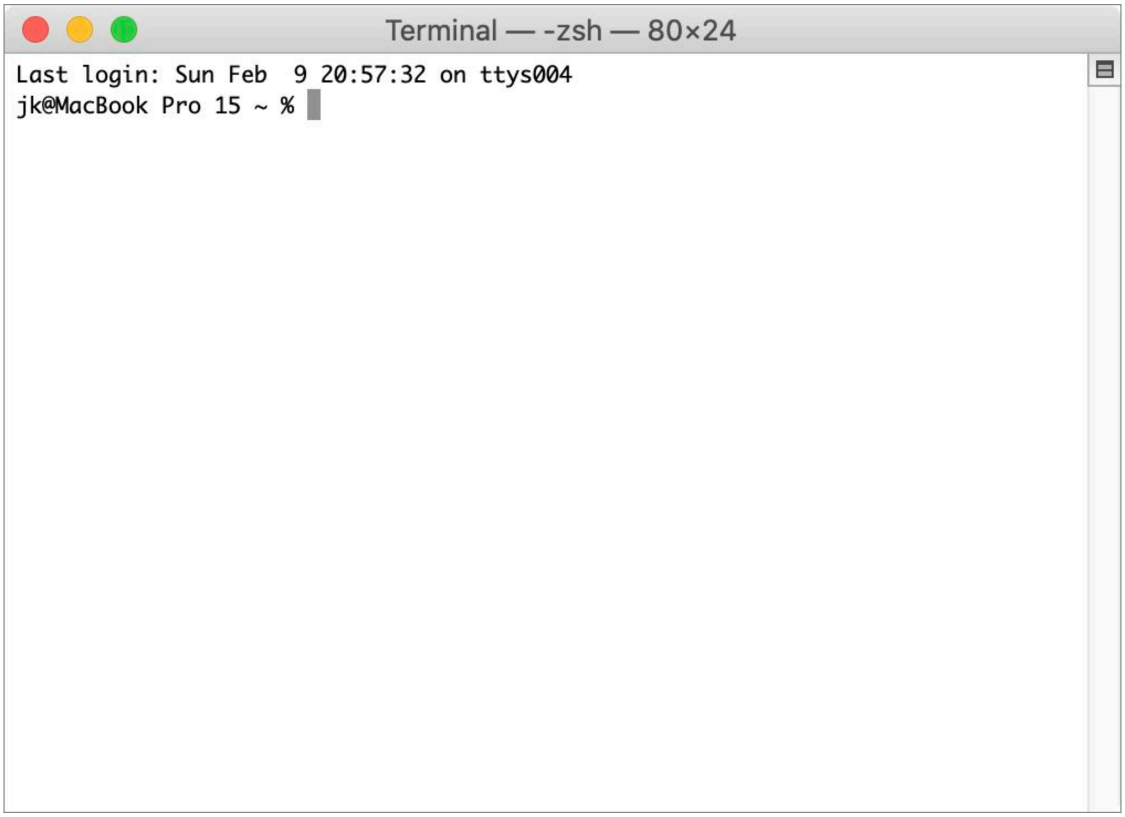
After another moment, a window appears with various tools, and you can continue with whatever tasks you want to perform.

Get to Know (and Customize) Terminal

As I mentioned in What's Terminal?, the app you're most likely to use for accessing the command line in macOS is Terminal. Since you'll be spending so much time in this app, a brief tour is in order. In addition, you may want to adjust a few settings, such as window size, color, and font, to whatever you find most comfortable and easy to read.

Learn the Basics of Terminal

The moment has arrived. Find the Terminal app (inside the folder `/Applications/Utilities`), double-click it, and take a Zen moment to contemplate the emptiness (**Figure 2**). The exact appearance of the window will vary based on your version of macOS.



```
Last login: Sun Feb  9 20:57:32 on ttys004
jk@MacBook Pro 15 ~ %
```

Figure 2: The Terminal window harks back to pre-graphical days.

To state the obvious, it's a (mostly) empty window. A Terminal window simply shows a command-line interface generated by a shell (in this case, the `zsh` shell). As long as you're in this window, you can largely forget about your mouse or trackpad: with a couple of notable exceptions (see the sidebar Using a Mouse in Terminal), everything you do here uses the keyboard only.

Of course, the window isn't *completely* empty. The first line lists, by default, the date and time of your last login. In this example, it's:

```
Last login: Sun Feb  9 20:57:32 on ttys004
```

That last part, `on ttys004`, is a bit of esoteric information that signifies the terminal interface with which you logged in the last time. It might say something different (such as `on console`) or nothing at all—for all practical purposes, you can safely ignore this line.

The second line is the actual command line (the line on which you type commands):

```
jk@MacBook Pro 15 ~ % ■
```

If you're using `bash` rather than `zsh`, it may look more like this:

```
MacBook-Pro-15:~ jk$ ■
```

Either way, the rectangular box at the end (which may instead appear as a vertical line or an underscore, any of which may or may not blink) is the *cursor* (not to be confused with the *pointer*, which reflects mouse movement). Everything before the cursor is known as the *prompt*, which is to say it's prompting you to type something.

The first part of the prompt, `jk@MacBook Pro 15` in the case of the `zsh` shell, is my short username followed by the @ sign and the name of my Mac. Next is the tilde (`~`), which signifies that I'm currently in my home directory (which, for me, is `/Users/jk`). And finally, the `%` signifies that I'm logged in as an ordinary (non-root) user. (I say more about the `%` and the `bash` equivalent, `$`, in the sidebar The %, $, and # Symbols on My Command Line, ahead.) If your short username is cindy and your computer's name, as shown in System Settings >

General > About or System Preferences > Sharing, is Cindy’s Groovy iMac, your command line may look something like this:

```
cindy@Cindy's Groovy iMac ~ % █
```

All these things are customizable; see Customize Your Defaults.

Tip: To learn a few advanced Terminal tricks, see Terminal Tips and Shortcuts.

Using a Mouse in Terminal

Although you’re never *required* to use a mouse or trackpad in Terminal—and all command-line programs were designed to be used with only a keyboard—there are a few situations in which a pointing device can come in handy:

- You can use your mouse to select text (for copying, say), just as you would in any other Mac app.
- You can drag a file or folder in from the Finder to copy its path to the command line, formatted in such a way that you don’t have to worry about spaces or other reserved characters (read Get the Path of a File or Folder).
- In the `nano` text editor, you can Option-click to move your cursor to that spot (or the nearest valid location). This also works on the current command line; for example, if you paste in a long command but want to make a change near the beginning, you can Option-click at that point, delete some characters, and type something new.
- You can ⌘-double-click a URL on the command line to open it in your default browser.
- Starting in 10.10 Yosemite, you can scroll (for example, using a two-finger vertical swipe on a trackpad, or with a scroll wheel on a mouse) through `man` (“manual”) pages, and move the cursor up or down by line (just as if you pressed ↑ or ↓ repeatedly) in programs such as the `nano` text editor.

Modify the Window

The window you're looking at is just like any other macOS window. You can move it, minimize it, resize it, zoom it, scroll through its contents, and hide it using the usual controls. So please do adjust it to your liking. However, I want to make two important points about window modification:

- First, resizing isn't only a good idea, it's practically mandatory. Some commands you run in this window generate a lot of text, including some large tables, and you'll find it much easier to work in the command line if your Terminal window is a bit bigger. Go ahead and make the window as large as you want—but do leave at least a bit of space so that you can see some parts of other windows on your screen.
- Second, any changes you make to the window ordinarily last only until you close it. If you open a new window—or quit Terminal and launch it again later—you're returned to the defaults. So, once you get your Terminal window to a size, shape, and position you like, choose Shell > Use Settings as Default. Thereafter, all new Terminal windows that you open use your preferred characteristics. (I say more about customizing windows ahead, in Change the Window's Attributes.)

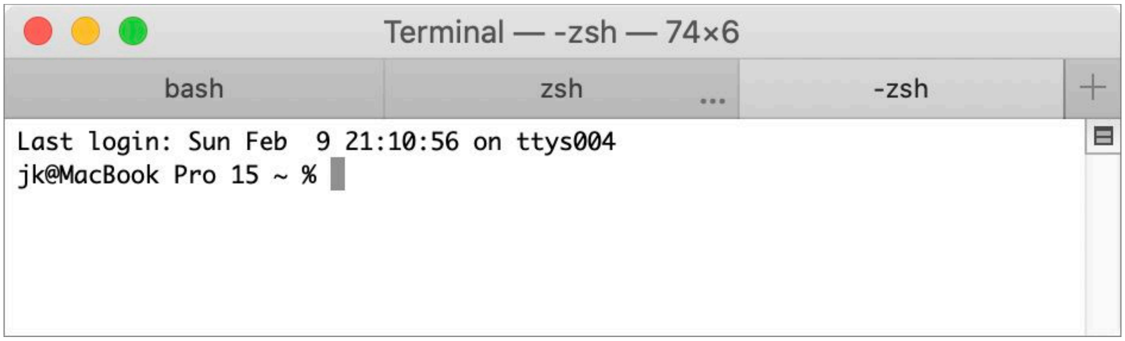
Open Multiple Sessions

Most apps can have multiple windows open at once—think of your word processor, your web browser, or your email program, for example. The same is true of Terminal—you can have as many windows open as you need, each with its own command line. To open a new window, press ⌘-N.

When you open a new window in Terminal, you begin a new *session*. That means another copy of the shell runs, separate from the first copy. You can run a program or navigate to a location in the first session, and run a completely different program or navigate to another location in the second. The two sessions don't normally interact at all; it's as though you're using two different computers at once that happen to share the same set of files.

Why would you want to do this? Perhaps you want to refer to a program's `man` (manual) page in one window, while trying out the command in a second. Perhaps one shell is busy performing some lengthy task and you want to do something else at the same time. Or perhaps you want to compare the contents of two directories side by side. Whatever the case, remember: you're not limited to using one window—or one session—at a time.

But wait, there's more! Every Terminal window supports multiple tabs—just like most web browsers (**Figure 3**). So if you want to have multiple sessions open without the screen clutter of multiple windows, you can do so easily. Create a new tab by pressing ⌘-T. Exactly as in a browser, you can drag tabs to rearrange them, close them individually, and even drag a tab from one window to another.



```
Last login: Sun Feb  9 21:10:56 on ttys004
jk@MacBook Pro 15 ~ %
```

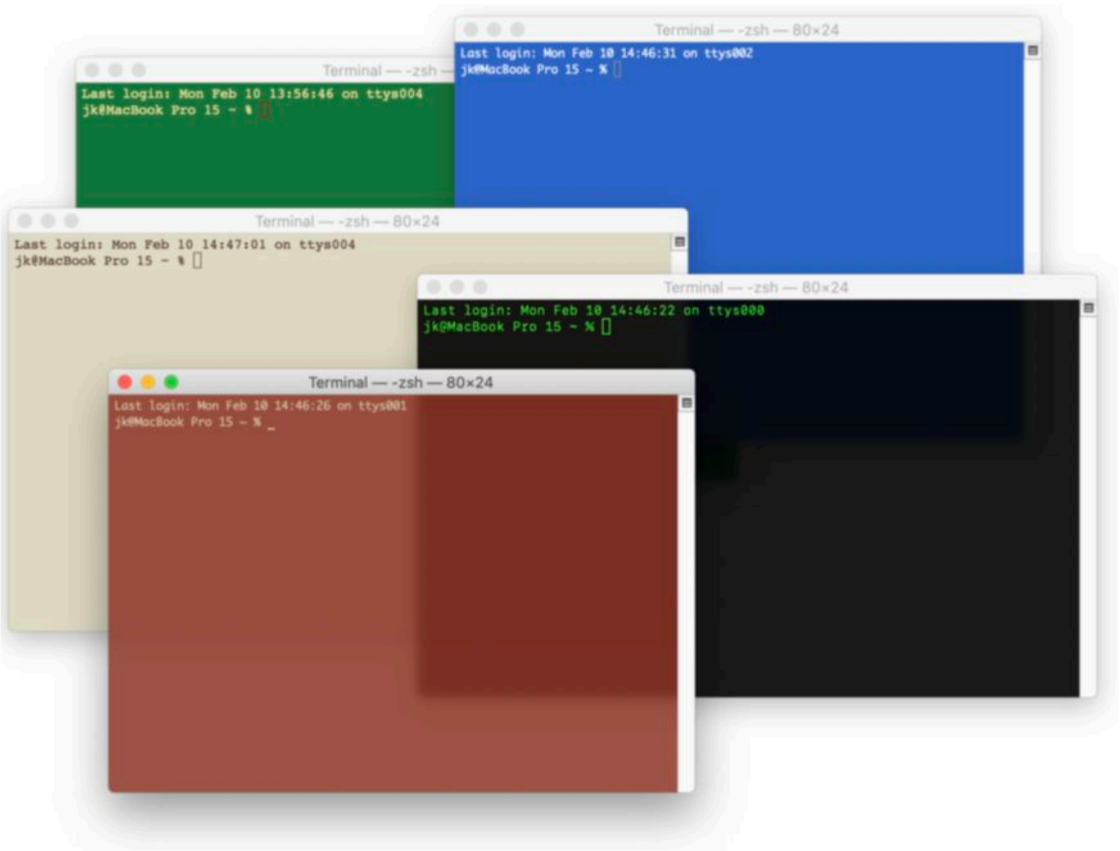
Figure 3: Terminal windows can have multiple tabs, which can be moved and closed individually just like those in most web browsers.

Change the Window's Attributes

Moving and resizing windows is one thing, but Terminal lets you go further. You can change the background color (and transparency), font (typeface and size), text color, cursor type, and numerous other settings. In fact, you can change far more attributes than I care to describe here, so I want to explain just a few of the basics.

For starters—just to get a feel for what's possible—choose Shell > New Window (or New Tab) and try some of the prebuilt profiles (previously called themes). For example, choose Shell > New Window > Homebrew for a display with bright green text in 12-point Andale Mono against a slightly transparent black background. Or choose Shell > New Window > Grass for pale yellow text, in bold 12-point Courier, on an opaque green background, with a red cursor.

Figure 4 shows several examples.



```
Last login: Mon Feb 10 14:46:31 on ttys002
jk@MacBook Pro 15 ~ %
```

```
Last login: Mon Feb 10 13:56:46 on ttys004
jk@MacBook Pro 15 ~ %
```

```
Last login: Mon Feb 10 14:47:01 on ttys004
jk@MacBook Pro 15 ~ %
```

```
Last login: Mon Feb 10 14:46:22 on ttys000
jk@MacBook Pro 15 ~ %
```

```
Last login: Mon Feb 10 14:46:26 on ttys001
jk@MacBook Pro 15 ~ % _
```

Figure 4: Terminal windows can take on many appearances; this image shows several of the stock profiles. (The exact appearance depends on which version of macOS you're running.)

If you prefer to use one of these other profiles as your default, open a new window with that theme and choose Shell > Use Settings as Default. But you can also modify these profiles or create your own.

To modify your window's appearance, follow these steps:

1. Choose Terminal > Settings/Preferences > Profiles. (In much older versions of macOS, go to Settings instead of Profiles.)

2. Select a profile in the list to modify it. Or, to create your own new theme based on an existing one, select a theme and choose Duplicate Settings from the pop-up action ⚙⌄ menu at the bottom of the list—or click the plus + button to add your own theme from scratch.

3. To modify the text and colors that appear in the window of the currently selected theme, click Text. A few of the more useful options in this view are the following:

    - **Background color:** Click the Color & Effects button under Background to open the Colors palette, in which you can choose a background color for the window. (Note: This setting was on the Window pane in much older versions of macOS.) You can also adjust the opacity of the background color. Why would you want a partially transparent window? I like transparency because I can put a Terminal window directly above, say, a webpage and read instructions through the window as I type in Terminal! To adjust the opacity, move the Opacity slider at the bottom of the Colors palette.

    - **Font:** To change the typeface or size, click the Change button, select a new font, size, and style from the Fonts palette, and close the palette. For best results, I strongly recommend choosing a fixed-width (monospaced) font, such as Courier, Monaco, Lucida Console, or SF Mono.

    - **Text color:** To change the color of the font, click the color button to the left of the word Text and chose a color using the Colors palette. You can pick a separate color for boldface text and for text you've selected with the mouse by clicking the color buttons next to Bold Text and Selection, respectively.

    - **Cursor attributes:** To change the shape of the cursor, select the Block, Underline, or Vertical Bar radio button. Check Blink Cursor if you want it to blink, and if you want to change the cursor's color, click the color button next to the word Cursor.

4. To modify the window itself, click Window. Some options you can change here include:

   - **Title bar elements:** To change the name of the window ("Terminal" by default), type new text into the Title field. You can also select any or all of the checkboxes beneath to display other information in the title bar, such as the name of the active process or the window's dimensions. (Terminal windows express their size in terms of rows and columns of text rather than in pixels. By default, Terminal windows are 24 rows by 80 columns, a size that harks back to old-style text-only terminals.)

   - **Window size:** You can change the default window size for the current profile by typing numbers into the Columns and Rows fields, or you can simply resize the window to your liking later by dragging the resize control at the window's lower-right corner.

Note: My personal preference for window appearance is white text on a black background, with a larger window (160 columns by 50 rows) and background transparency set to 85%.

5. To make a particular profile the default (which means it's used automatically when you launch Terminal, and when you press ⌘-N), select it and click the Default button beneath the list of themes. When you're finished adjusting window settings, close the Settings window.

The settings you change here *should* take effect immediately for existing windows using the selected profile, and definitely take effect for the next new window or tab opened using that profile.

Change Your Current Shell

A shell is a command-line program, and you can run a shell the same way you run any other program in Terminal (something I cover in more detail in Run a Program or Script). So, if you want to use a different shell than the one that ran automatically when you opened your current Terminal session, all you have to do is enter its name (for example, `zsh` or `bash`).

This assumes both default settings and one of Apple's six or seven (depending on your version of macOS) built-in shells: `bash`, `csh`, `ksh`, `sh`, `tcsh`, `zsh`, or (starting in Catalina) `dash`. If you've changed your PATH (see Modify Your PATH), installed an alternative shell, or made any other unexpected configuration changes, you might have to enter the full pathname to the shell (for example, `/bin/zsh`—all the shells included with macOS are in `/bin`).

If you've switched shells within the current session and want to go back to the one you were running previously, enter `exit`.

Set a Default Shell

As I explained in the introduction, this book covers only the `zsh` and `bash` shells, and regardless of which version of macOS you're running, *either one of those shells is just fine*. But if you're unsure which shell you're running or want to switch to a different one, here's how.

Find Out Which Shell You're Using

To find out which shell is currently running, enter this:

```
echo $0
```

The shell replies with its own name, sometimes preceded by a hyphen:

```
zsh
```

or

```
-bash
```

Change Your Default Shell

If you want to change the default shell for your user account, there are three ways to do so—one that uses a command-line program, one that uses System Settings/System Preferences, and a third that uses Terminal's Settings/Preferences.

Change Your Default Shell from the Command Line

To change your default shell on the command line using the `chsh` ("change shell") command:

1. Enter `chsh -s /bin/shell` (where *shell* is replaced with the shell you want to use as your default). For example:
   ```
   chsh -s /bin/zsh
   ```

2. Enter your password when prompted to do so.

That's it—you've changed your default. Note, however, that changing your default doesn't change your *current* shell; the change takes place when you next open a new Terminal tab or window. If you want to switch to the new shell immediately, just enter its name (like `zsh`).

Change Your Default Shell in System Settings/System Preferences

To change your default shell using System Settings/System Preferences:

1. Open System Settings/System Preferences > Users & Groups. (If you're using Lion or earlier, open System Preferences > Accounts.)

2. In Monterey or earlier, if the lock 🔒 icon in the lower left of the window is closed, click it and enter your administrator's credentials to authenticate.

3. Right-click (or Control-click) on your name in the list, and choose Advanced Options from the contextual menu. In Ventura or later, enter your administrator's credentials to authenticate.

4. In the dialog that appears, choose a different shell from the Login Shell pop-up menu.

5. Click OK, and then close System Settings/System Preferences.

Although the Advanced Options pane may warn that you need to restart your computer to apply changes, changing the default shell takes effect with the next Terminal session you open.

> **Note:** The above two methods change the default shell for your user account even when you access it via an app other then Terminal. To change your default shell *only* for Terminal, use the next method.

Change Your Default Shell in Terminal Settings/ Preferences

To change your default shell using Terminal's Settings/Preferences:

1. Choose Terminal > Settings/Preferences > General.
2. Next to "Shells open with," select "Command (complete path)" and make sure the path to `zsh` (`/bin/zsh`) is filled in. (To use a different shell, such as `bash`, substitute that shell's name for `zsh`.)

The setting—which *overrides* changes made with either of the above methods but affects only Terminal—applies with your next session.

The %, $, and # Symbols on My Command Line

By default, when you open a Terminal window, you see a prompt that ends in a `%` (for `zsh`) or a `$` (for `bash`), followed by the cursor, like so:

```
jk@MacBook Pro ~ % ■
```

or

```
MacBook-Pro:~ jk$ ■
```

(Note that zsh shows your short username at the beginning, and includes spaces in your computer name; bash puts your username at the end and replaces spaces with hyphens. Other shells may have different default prompts.)

If you log in as the root user (see Perform Actions as the Root User), the prompt ends instead in a `#` character:

```
root@MacBook Pro ~ # ■
```

or

```
bash-3.2# ■
```

As a result, when you're reading articles and websites listing commands you might enter in Terminal, you might run across examples like these:

```
$ open -e file1
```

```
% top
```

```
# chown www file1
```

The %, $, or # at the beginning merely signifies that what follows is a command to be typed and, in the case of `#`, that it's supposed to be typed by the root user. You wouldn't actually type `$`, `#`, or `%`.

I don't use that convention in this book; whatever you need to type on the command line simply appears in a special font, usually on a line by itself. I find those extra characters distracting.

In any case, you can easily change the prompt so that it shows something else entirely. If you want your prompt to look like this…

```
Joe rocks +> ■
```

…you can make that happen. See Change Your Prompt for details.

Look Around

In this chapter, I help you find your way around your Mac from the command line and, at the same time, teach you some of the most common navigational commands and conventions.

For right now, you're going to look, but not touch—that is, nothing you do here can change any files or cause any damage, as long as you follow my instructions.

Discover Where You Are

Ready to start learning some commands? Here we go. Open a Terminal window and enter this:

```
pwd
```

Note: As a reminder, to *enter* something on the command line, type it and press Return or Enter afterward.

The `pwd` command stands for "print working directory," and it gives you the complete path to the directory you're currently using. If you haven't done anything else since opening a Terminal window, that's your home directory, so you'll see something like this:

```
/Users/jk
```

That's not exciting, but it's extremely important. As you navigate through the directory structure, it's easy to get lost, and ordinarily your prompt only tells you the name of your current directory, not where it's located on your disk. When you're in a deeply embedded directory, being able to tell exactly where you are can be a huge help.

Note: When I say "disk" in this book, I'm referring to any mass-storage device, not necessarily a mechanical hard disk. It could be an SSD or other flash storage.

See What’s Here

If you were in the Finder, you’d know exactly what’s in the current folder just by looking. Not so on the command line; you must ask explicitly. To get a list, you use the “list” command:

```
ls
```

What you get by default is a list along the lines of the following:

```
Desktop      Downloads     Movies       Pictures
Documents    Library       Music        Public
```

Items are listed alphabetically, top to bottom and then left to right. But as you can see, this doesn’t tell you whether these items are files or directories, how large they are, or anything else about them. So most people prefer the more-helpful long format by adding the `-l` flag:

```
ls -l
```

This produces a result something like:

```
drwx------@  18 jk    staff    612 Feb 12 09:42 Desktop
drwx------@ 108 jk    staff   3672 Feb  9 14:35 Documents
drwx------@  15 jk    staff    510 Feb 12 11:17 Downloads
drwx------+  94 jk    staff   3196 Feb 11 22:40 Library
drwx------@  13 jk    staff    442 Dec 30 15:34 Movies
drwx------@  15 jk    staff    510 Aug 27 15:02 Music
drwx------+  14 jk    staff    476 Jan 26 19:40 Pictures
drwxr-xr-x+   7 jk    staff    238 Jan 22 23:13 Public
```

Reading from right to left, notice that each line ends with the item’s name. To the left of the name is a date and time showing when that item was most recently modified. To the left of the date is another number showing the item’s size in bytes. See the sidebar on the next page, Making Output (More) Human-Readable, to find out how to turn that number into a nicer format. (In the case of a directory, the number shown by `ls -l` doesn’t tell you the total size of the directory’s *contents*, only the size of the information stored *about* the directory.

The “disk usage” command can get a directory’s size, enter `du -sh directory-name`.)

Later in this book, in Understand Permission Basics, I go into more detail about all those characters that occupy the first half of each line, such as `drwxr-xr-x +   7 jk     staff`; those characters describe the item’s permissions, owner, and group. For the moment, just notice the very first letter—it’s `d` in every item of this list. The `d` stands for “directory,” meaning these are all directories. If the item were a file, the `d` would be replaced with a hyphen (`-`), for example: `-rwxr-xr-x`.

Finally, look at one other number, between the permissions and owner (in `drwx------    14 jk` the number is 14). That’s the number of *links* to the item, and although links are too advanced to explain in detail here, the number serves one practical purpose: it gives you an approximation of the number of items in a directory. In fact, it is always at least two higher than the number of visible files or directories in the directory (for complicated reasons). For now, just know that the number can tell you, at a glance, if a directory has only a few items or many.

Making Output (More) Human-Readable

I’ve shown the `-l` (long format) flag, which provides much more detail than the `ls` command alone. But it shows the file size in bytes, which isn’t a convenient way to tell the size of large files. For example, an `ls -l` listing might include the following:

```
-rw-r--r--@  1 jk  admin  15132784 Jan 13 17:07 image.dmg
```

Really—15132784 bytes? Wait a minute, let me do some math…how large is that exactly?

Luckily, you can improve on this by adding the `-h` flag, which stands for “human-readable.” (In fact, `-h` works with many commands, not just `ls`.) You can enter either ls `-lh` or `ls -l -h`. Either way, you get something like this:

```
-rw-r--r--@  1 jk  admin  14M Jan 13 17:07 image.dmg
```

Aha! The file is 14 megabytes (M) in size. That I understand!

However, as I hinted in The Startup Volume Became Read-Only, starting in Catalina, the `ls` command doesn’t always tell the entire story. For example, if you enter `ls /Applications` you'll see *only* the third-party applications (which are really stored in `Macintosh HD - Data/Applications`). What if you want to see the apps built into macOS—the ones that are stored in `Macintosh HD/Applications`? In that case, you have to enter `ls /System/Applications`. (The same goes for `/Library` and other top-level directories.) You can't use the `ls` command to show you the merged contents of these folders as the Finder does.

> **Note:** What's true for `ls` applies to other commands too—if you want to perform any operations on files or directories that are part of the read-only system volume, you must include `/System` before the rest of the path as it appears in the Finder.

I don't want to belabor the `ls` command, but it will without question be one of the top two or three things you type on the command line—you'll use it constantly. So it pays to start getting `ls` (along with a flag or two) into your muscle memory. For a way to display even more information with `ls`, see the recipe List More Directory Information.

> **Note:** You can also list the contents of a directory other than your current one like this: `ls /some/other/path`.

Repeat a Command

If you've just entered a two-character command, it's no big deal to enter it again. But sometimes commands are quite complex, wrapping over several lines, and retyping all that is a pain. So I want to tell you about two ways of repeating commands you've previously entered.

Arrow Keys

First, you can use the ↑ and ↓ keys to move backward and forward through the list of commands you've recently typed. For example, if the last command you typed was `ls -lh`, simply pressing ↑ once puts that on the command line. (Then, to execute it, you would press Return or Enter.) Keep pressing ↑, and you'll step backward through even more commands. You can even scroll through commands you entered *in previous sessions*. The ↓ key works the same way—it progresses forward in time from your current location in the list of previous commands.

Tip: To learn another useful way to access your command history, see Search Your Command History.

The !! Command

Another handy way of repeating a command is to enter `!!` (that's right: just two exclamation points). This repeats your previous command. Try it now. Enter, say, `pwd`, and get the path of your current directory. Then enter `!!` and you'll get the same output.

Again, this isn't terribly interesting when you're talking about short commands, but it can save time and effort with long commands.

!! Plus

The `!!` need not stand alone on the command line—you can add stuff before or after in order to expand the previous command. For example, if you previously entered `ls -l` and you now want to enter `ls -l -h`, you could repeat the previous command and add an extra flag like this:

```
!! -h
```

Or, if you enter a command like `rm file1` (remove the file `file1`) and get an error message telling you that you don't have permission, you can repeat it preceded by the `sudo` command (described in Perform Actions as the Root User):

```
sudo !!
```

In this example, the result would be exactly the same as entering:

```
sudo rm file1
```

But wait, there's more! You can use `!$` to repeat only *arguments* from the previous command. So, if you used `ls -la /etc/ssh/ssh_config.d` to list the contents of that directory, and you decide you want to switch to it, you could then enter `cd !$`.

Finding Text in the Terminal Window

As you work in Terminal, the output of earlier commands (such as file lists) scrolls upward, and you can easily accumulate many thousands of lines of output in a single session.

To find some text within your current Terminal session (without manually scrolling and looking for it), you can press ⌘-F (for "Find"), type a search term, and if necessary press ⌘-G (for "Find Next") to go to the next instance or ⌘-Shift-G to go to the previous instance.

Apple enhanced Terminal's Find command starting in Yosemite by making it inline (with Previous and Next buttons at the top of the window), as opposed to a Find dialog used in earlier versions of macOS.

Cancel a Command

What if you type some stuff on the command line and realize you don't want to enter the command? Well, you *could* backspace over it, but that may take a while if there's a lot of text on the line.

An easier way to back out of a command without executing it is to press either Control-C or ⌘-. (period). The shell creates a new, blank command line, leaving your partially typed line visible but unused. (Your command history won't include canceled commands.)

Move into Another Directory

This has been a lovely visit in your home directory, but now it's time to explore. To change directories, you use the `cd` command. As you saw a moment ago, one of the directories inside your home directory is called `Library`. Let's move there now, like so:

```
cd Library
```

Note: Notice that `Library` is capitalized here. Sometimes case isn't important on the command line (as I explain ahead in Case Sensitivity), but you can't go wrong if you always use the correct case.

When you put a directory name after the `cd` command, it assumes you want to move into that directory *in your current location*. If there doesn't happen to be a directory called `Library` in your current directory, you see an error message like this (preceded by `-bash` if you're using the `bash` shell):

```
cd: Library: No such file or directory
```

As a reminder, the command-line environment doesn't list the contents of a directory unless you ask it to (using `ls`), so using `cd` doesn't automatically show what's in your new location. You know the command succeeded if you don't see an error message, and by default your prompt includes the name of your current directory.

Move Up or Down

Now that you're in the `Library` directory that's in your home directory (`~/Library`), you can use `ls` to look around; you'll see that one of the directories inside the current one is `Preferences`. To move down a level into preferences, you'd enter `cd Preferences`. And so on.

To go up a level, you use the `..` convention, which means "the directory that encloses this one." For example, if you're in `/Users/jk/Library/Preferences`, the directory that encloses it is `/Users/jk/Library`, so in this particular location two periods (`..`) means `/Users/jk/Library`.

To get there, you enter:

```
cd ..
```

That translates as "change directories to the one that encloses this one." You can keep going up and down with `cd ..` and `cd directory` (fill in the name of any directory) as much as you like.

Note: Moving into directories with spaces in their names requires extra effort; read Understand How Paths Work, ahead.

Move More Than One Level

Nothing says you have to move up or down just one level at a time. If you're currently in `/Users/jk` and you know that there's a `Library` directory inside it, and inside that there's a `Preferences` directory, you can jump directly to `Preferences` like so:

```
cd Library/Preferences
```

The slash (`/`) simply denotes that the term to its right is a directory inside the term on its left: `Preferences` is a directory inside `Library`. You can add on as many of these as you need:

```
cd Library/Logs/Adobe/Installers
```

This also works in the other direction. If you're in `/Users/jk/Library/Preferences`, you can enter `cd ..` to move into `Library`. Or, enter `cd ../..` to move directly into `jk`, or `cd ../../..` to move into `Users`.

Move to an Exact Location

So far, we've been moving using relative locations—a directory inside the current one, or a directory that encloses the current one. But if you know exactly where you're going, you can jump directly to any location on your Mac. Just specify the full path, beginning with a slash (`/`), which represents the root level of your disk. For example, enter this:

```
cd /private/var/log
```

That takes you directly to `/private/var/log` (a rather boring directory full of—you guessed it—log files, and one that's normally invisible in the Finder) without having to navigate all the way up to the root level of your disk and then back down.

Speaking of the root level: If you want to go to the very top of your disk hierarchy, just enter this:

```
cd /
```

Note: As a reminder, in Catalina or later, if you want to go to a specific location on the read-only system volume, you must put `/System` before the rest of the pathname—for example, `cd /System/Developer`.

Move Between Two Directories

Another handy shortcut, which lets you go back to the last directory you were in, is this:

```
cd -
```

For example, suppose I start in my home directory and then I enter `cd /Users/Shared`. I do some things in that directory, and I next enter `cd ~/Library/Preferences` to do other work there. If I then enter `cd -` I jump back to `/Users/Shared` (the last directory I was in), without having to type or even remember its path.

Jump Home

Once you've changed directories a few times, you may want to get back to your home directory. Of course, you could keep navigating up or down, one directory at a time, until you got there, or you could enter the complete path to your home directory (`cd /Users/jk`, for example). But macOS has another shortcut (along the lines of `..`) that means "the current user's home directory": the tilde (`~`).

So one way to jump home, from any location on your disk, is to enter:

```
cd ~
```

But in fact, it can be even easier. If you enter `cd` alone, with nothing after it, the command assumes you want to go home, so `cd` by itself does the same thing as `cd ~`.

Just as you can enter the full path after `cd` to jump to any spot on your disk, you can substitute `~` whenever you'd otherwise use the full path to your home directory. So, even if you're in `/private/var/tmp`, you can go directly to the Library directory inside your home directory with:

```
cd ~/Library
```

Note: This might be a good time to remind you that the command line can be unforgiving. If you type an extra period, leave out a space, or make some other similarly tiny error, your command might not work at all—or it might do something entirely unexpected. That need not frighten you, but be aware that you should be deliberate and careful when typing on the command line.

Understand How Paths Work

You've already seen both relative paths (such as `Library/Preferences`, which means the `Preferences` directory inside the `Library` directory inside my *current* directory) and absolute paths, which begin with a slash (such as `/Library/Preferences`, which means the `Preferences` directory inside the `Library` directory at the *top* level of your disk). But there are a few other things you should understand about paths.

Spaces in Paths

macOS lets you put almost any character in a file or folder name, including spaces. But space characters can get you in trouble in the command-line environment, because normally a space separates commands, flags, and arguments.

Suppose you were to enter this:

```
cd My Folder
```

Even if there were a folder named `My Folder` in the current directory, the command would produce an error message, because the `cd` command would assume that both `My` and `Folder` were intended to be separate arguments.

You can deal with spaces in either of two ways:

- **Quotation marks:** One way is to put the entire path in quotation marks. For example, entering `cd "My Folder"` would work fine.
- **Escape the space:** The other way is to put a backslash (`\`) before the space—this *escapes* the space character, making the shell treat it literally rather than as a separator between arguments. So this would also work: `cd My\ Folder`.

Note: To be crystal clear, the backslash (`\`) is normally located on a key just to the right of the] key on a U.S. Macintosh keyboard. It has a completely different meaning from the ordinary (forward) slash (`/`), located on the same key as the question mark. Don't mix them up!

Terminal automatically escapes the name of a file or folder when you drag it in from the Finder. See Get the Path of a File or Folder, later.

Wildcards

You can use wildcards when working on the command line; these can save you a lot of typing and make certain operations considerably easier. The two wildcards you're most likely to use are these:

- *** (asterisk):** This means "zero or more characters." For example, if you want to switch to a directory called `Applications`, you could enter `cd App*` and, as long as there was no other directory there that started with those three letters, you'd go directly to the `Applications` directory. (I talk about another way of doing something similar ahead a few pages in Use Tab Completion.)

  You can use this wildcard with almost any command. For instance, if you're in your home directory, you could type `ls D*` to list all and only the items that begin with "D" (`Desktop`, `Documents`, `Downloads`).

- **? (question mark):** This means "any single character." That means `?at` could match `bat`, `cat`, `fat`, `rat`, `sat`, and so on. If you have many files with similar names—say, sequentially numbered photos—you could limit the ones listed with something like `ls 01??.jpeg`.

Case Sensitivity

Here's a trick question: is the macOS command line case-sensitive? The answer is yes—and no! Suppose you're in `~`. There's a directory in there called `Pictures`, and you could move into it in any of these ways (among others):

```
cd Pictures
```

```
cd pictures
```

```
cd Pic*
```

```
cd pic*
```

That certainly seems to suggest that the command line is *not* case-sensitive, because using either p or P has the same effect. But it's possible to format a Mac volume to use a case-sensitive version of the APFS or Mac OS Extended (HFS+) file systems. If you do that—or if you connect to an external disk or network volume that uses a case-sensitive file system—then you could see both a `pictures` directory and a `Pictures` directory in the same place, in which case using the wrong case with the `cd` command will take you to the wrong directory.

You won't see any visual cue to let you know whether a particular volume uses a case-sensitive format. So the safest assumption is to always use the correct case: that always works.

Understand the macOS Directory Structure

You surely know from day-to-day use that your Mac has a bunch of standard folders at the top level of your startup disk—`Applications`, `Library`, `System`, and `Users`, at minimum. You may have also noticed that each user's home folder has its own `Library` folder (not to mention a `Desktop` folder, a `Documents` folder, and several others). In addition to these and the numerous other folders you can see in the Finder, macOS has a long list of directories that are normally invisible (because most users never need to interact with them directly), but you can see them from the command line.

I could explain what every single (visible) folder and (hidden) directory is for, and how to make sense of the elaborate hierarchy in which macOS stores all its files. But that would take many pages and, honestly, it would be mighty boring. So I'm going to let you in on a little secret: *you don't have to know*.

I mean it: you don't have to know why one program is stored in `/bin` while others are in `/usr/bin`, `/usr/local/bin`, or any of numerous other places. You don't need to know why you have a `/dev` directory or what goes in `/private/var`. Seriously. Knowing all those things might be useful if you're a programmer or a system administrator, but it's absolutely irrelevant for ordinary folks who want to do the kinds of things discussed in this book. True, I may direct you to use a program in `/usr/sbin` or modify a file in `/private/etc` (or whatever), but as long as you can follow the instructions to do these things, you truly don't have to know all the details about these directories.

So, instead, I want to provide a very short list of the key things you should understand about the macOS directory structure:

- **The invisible world of Unix:** If you enter `ls -l /` (go ahead and do that), you get a list of all the files and directories at the root level of your disk. (In Catalina or later, this combines items from the read-only system volume and the writable Data volume.) You'll see familiar names such as `Applications` and `Users`, and some less-familiar ones, such as `bin` and `usr`. Here at the root level, directories that begin with a lowercase letter and aren't shown in the Finder (such as `bin`, `cores`, `dev`, `private`, `sbin`, `usr`, and `var`), along with their contents, make up Darwin, the Unix core of macOS. Similar directories appear in other Unix and Unix-like operating systems.
- **Recursion, repetition, and recursion:** If you were to work your way from the root of your disk down through all its directories and subdirectories, you'd notice a lot of names that appear over and over again. For example, there's a top-level `/Library` directory, another inside `/System`, and yet another inside each user's home

directory (`~/Library`). Similarly, there are top-level `/bin` and `/sbin` directories, but also `/usr/bin` and `/usr/sbin`. The reasons for all these copies of similar-looking directories are sometimes practical, sometimes purely historical. But everything has its place.

  You don't have to grasp all the logic behind what goes where, but you do have to be sure you're in the right place when you work on the command line. For instance, if an example in this book tells you to do something in `~/Library`, be absolutely sure that's where you are, as opposed to, say, `/Library`. The smallest characters—in particular, the period (`.`), tilde (`~`), slash (`/`), backslash (`\`), and space (` `), have the utmost significance on the command line, so always pay strict attention to them!

- **The bandbox rule:** My grandfather had a curious and oft-repeated expression: "Don't monkey with the bandbox." He (and, subsequently, my mother) used this to mean, approximately, "Don't mess with something if you could break it and not be able to put it back together." (As a child, I had quite a propensity for disassembling things and then getting stuck!)

  On the command line, this means don't go deleting, moving, or changing files if you don't know what they are or what the consequences could be. Something that seems insignificant or useless to you could be crucial to the functioning of your Mac. (As a corollary, it should go without saying that you back up your Mac thoroughly and regularly—see my book *Take Control of Backing Up Your Mac* if you want detailed advice on the subject.)

Use Tab Completion

Because everything you do on the command line involves typing, it can get kind of tedious spelling out file and directory names over and over again—especially since even the slightest typo can make a command fail! So the `zsh` and `bash` shells include a number of handy features to reduce the amount of typing you have to do. Earlier I explained how to use the arrow keys and the `!!` command to repeat previous commands

(Repeat a Command). Now I want to tell you about a different keystroke-saving technique: tab completion.

Here's the basic idea. You start typing a file or directory name, and then you press the Tab key. If only one item in the current directory starts with the letter(s) you typed, the shell fills in the rest of that item's name. If there's more than one match, you'll hear a beep; press Tab again to see a list of all the matches.

For example, try this:

```
cd
```

Now that you're in your home directory, type `cd De` (*without* pressing Return) and press Tab. Your command line should look like this:

```
cd Desktop/
```

If you do want to change to your desktop directory, you can simply press Return. Or, you can type more on the line if need be. For now, let's stay where we are—press Control-C to cancel the command.

Next, try typing `cd D` (again, without pressing Return) and press Tab. You should hear a beep—signifying that there was more than one match—but nothing else should happen. Press Tab again. Now you'll see something like this:

```
Desktop/    Documents/    Downloads/
```

And, on the previous line (in `zsh`) or next line (in `bash`), your command-in-progress appears again exactly as you left it off:

```
cd D
```

In this way, tab completion lets you know what your options are; you can type more letters (say, `oc`) and press Tab again to have it fill in `Documents/` for you. In `zsh`, you can keep pressing Tab to cycle through matching options, and press Return when the one you want appears.

Tab completion isn't limited to just the current directory. For example, enter `cd ~/Lib` and press Tab. The shell fills in the following:

```
cd ~/Library/
```

Now type `Fav` and press Tab. You should see `Favorites` filled in:

```
cd ~/Library/Favorites/
```

You can keep going as many levels deep as you need to.

> **Note:** Tab completion in `bash` is always case-sensitive, even on a volume that doesn't use case-sensitive formatting. If a directory is named Widgets, typing `wi` and pressing Tab produces no matches. In `zsh`, however, it's not case-sensitive unless you add `CASE_SENSITIVE="true"` to your `.zshrc` file; see About Startup Files.

Find a File

In the command-line environment, as in the Finder, you may not know where to find a particular file or directory. Two commands can supply that information readily: `find` and `locate`. Each has its pros and cons.

Find

To use the `find` command, you give it a name (or partial name) to look for and tell it where to start looking; the command then traverses every directory in the area you specify, looking at every single file until it finds a match. That makes it slow but thorough.

For example, suppose I want to find all the files anywhere in my home directory with names that contain the string *keychain*. I can do this:

```
find ~ -name "*keychain*"
```

After the command `find`, the `~` tells the command to begin looking in my home directory (and work its way through all its subdirectories). The `-name` flag says to look for patterns in the last item of a path (which may include the names of directories as well as files). I put the search string inside quotation marks, with an asterisk (`*`) wildcard at the beginning and end to signify that there may be other letters before or after *keychain*.

Even a simple search such as this one can take several minutes, because it must look at every single file, starting at the path I specified. To make it go quicker, I could specify a narrower search range. For example, to have it look only in my `~/Library` directory, I'd enter:

```
find ~/Library -name "*keychain*"
```

Let me offer a few other tips for using `find`:

- To search in the current directory (and all subdirectories), use a period (`.`) as the location: `find . -name "*keychain*"`.
- To search your entire disk, use a slash (`/`) as the location: `find / -name "*keychain*".`
- Normally, `find` is case-sensitive, so a search for `"*keychain*"` would *not* match a file named Keychain. To make a search case-insensitive, replace `-name` with `-iname`, as in `find ~ -iname "*user data*"`.
- During a search, if `find` encounters any directories you don't have permission to search, it displays the path of the directory with the message "Permission denied." To search these paths, use `sudo` before `find`, as described in Perform Actions as the Root User.

Tip: If you want to search the *contents* of files, you should instead use the `grep` command, though that process usually takes much longer. See how in Get a Grip on grep.

Locate

The other way to find files by name is to use the `locate` command. Unlike `find`, `locate` doesn't traverse every file to find what you're looking for. Instead, it relies on a database (index) of file and path names. The benefit of using the index is that `locate` is lightning fast. The downside is, the database is normally updated only once a week, so `locate` may be unable to find files you've added or renamed recently.

To use `locate`, just type that command followed by any portion of the filename you want to look for (no wildcards required). For example:

```
locate keychain
```

Like find, locate performs case-sensitive searches by default. To make a search case-insensitive, add the -i flag:

```
locate -i keychain
```

If you enter locate and get an error stating that no database exists—or if it exists but is outdated—you can create or update it by entering this:

```
/usr/libexec/locate.updatedb
```

The command may take some time to complete, because it does have to look at every file on your disk—or nearly so.

I've skipped over one important detail: by default, locate indexes (and finds) only files you own (mostly the contents of your home directory). However, if you run the database updating script using sudo (see Perform Actions as the Root User), it indexes every file on your disk, and locate can therefore find every file.

The benefit of this is being able to find more files with locate, but if you attempt to do this, a security warning appears informing you that once you've indexed all your files, any user of your Mac can discover the name and location (though not the contents) of any file on your disk. Moreover, the next time the locate database updates on its weekly schedule, your system-wide index of files will be replaced with a version that contains only those you have permission to read.

View a Text File

You may not read a lot of plain text files in the Finder, but the need to do so comes up more frequently in the command-line environment—reading documentation, examining programs' configurations, viewing shell scripts, inspecting logs, and numerous other situations. You can use many tools to read a file, of which I cover just a few here. (If you want to modify a text file, see Edit a Text File, later.)

You can use these commands with any text file on your Mac, but in these examples I use a file every Mac user should have: a dull Read Me file about OpenLDAP schema, located at /etc/openldap/schema/README.

More or Less

An early Unix program for reading text files was called `more`. It was pretty primitive and wouldn’t let you move backward to see earlier text. So a new program came along that was supposed to be the opposite of more: `less`. In macOS, both names still exist, but they point to the same program; whether you enter `more` or `less`, you’re actually running `less`. (There are some subtle differences depending on which command you use, but they’re not worth mentioning.)

You can use `less` to read a text file like this:

```
less /etc/openldap/schema/README
```

You see the top portion of the file initially. You can scroll down a line at a time using the ↓ key (and back up using the ↑ key), scroll ahead a screen at a time by pressing the Space bar, or backward a screen at a time by pressing the `B` key (all by itself). To quit `less`, press the `Q` key (all by itself).

Cat

The Unix `cat` command (short for “concatenate”) combines files, but you can also use it to display a text file on your screen. Unlike `less`, it doesn’t give you a paged view; it simply pours the entire contents of the file, regardless of length, onto your screen. You can then scroll the Terminal window up and down, as necessary, to view the contents. To use `cat`, follow this pattern:

```
cat /etc/openldap/schema/README
```

Tail

If you open a long text file with `less`, it can take quite a bit of tapping on the Space bar to reach the end, which is awkward if the information you want happens to be at the end—as is the case with most logs. And if you use `cat`, it can clutter your Terminal window with lots of information you don’t need. To jump to the end of a text file, use a different program: `tail`, which displays the *tail end* of a file.

If you enter `tail` followed by the filename, it displays the last 10 lines of the file:

```
tail /etc/openldap/schema/README
```

The `tail` command has flags that enable you to control how much of the file is shown and in what way, but for the sake of brevity I want to mention just one: `-n` (number of lines). Type `tail` followed by the `-n` flag, a space, and a number to set the output to that number of lines from the end of the file:

```
tail -n 50 /etc/openldap/schema/README
```

Head

Although the most interesting and useful stuff is more often at the end of a file, making `tail` the quickest way to view it, sometimes you may want to look at the beginning of a file instead. For that purpose, use the opposite program: `head`.

If you enter `head` followed by the filename, it displays the first 10 lines of the file:

```
head /etc/openldap/schema/README
```

To display a different number of lines, type `head` followed by the `-n` flag, a space, and a number:

```
head -n 50 /etc/openldap/schema/README
```

Get Help

Almost every program and command you use on the command line has documentation that explains its syntax and options, and in many cases includes examples of how to use the command. This documentation isn't always clear or helpful, but it's worth consulting when you have a question. You can get at these manual pages in several ways.

In a Terminal Window

When you're on the command line, the quickest way to get information about a command is to use the `man` ("manual") command. Simply enter `man` followed by the command you want to learn about. For example:

```
man ls
```

```
man cp
```

```
man locate
```

The results appear in a viewer that works like `less`.

Note: You can, of course, get instructions for using the man program itself by entering—you guessed it—`man man`.

To put a slightly prettier (and scrollable) display of `man` pages on the screen side by side with your working Terminal window, you can also click Terminal's Help menu, type the name of a command in the Search field, select the item you want, and press Return. If the command is already visible anywhere in the Terminal window, you can right-click (or Control-click) it and choose "Open man Page" from the contextual menu to view it in a separate window. Or, if your cursor happens to be immediately to the right of the term you want to see the man page for, simply press ⌘-Control-?.

Tip: If you want to read `man` pages as nicely formatted PDF files, try the recipe Read man Pages in Preview. Or, if you prefer to view them in BBEdit, try Read man Pages in BBEdit.

Tip: You can also read your Mac's man pages in your web browser using the free Bwana app. It hasn't been updated for many years, but it still appears to be mostly functional.

Clear the Screen

As you work in Terminal, your window may fill up with commands and their output. The command line itself is always the last line, but the rest of the window can become cluttered with the residue of earlier commands. Here are some ideas for decluttering the window:

- If you find all that text distracting and want to clear the window (so it looks much like it did when you started the session), enter `clear` or press Control-L. Terminal moves your command line up to the top of the window with empty space below it (you can still scroll up to see what was on the screen earlier).

- To clear the screen *and* prevent someone from scrolling back in Terminal to see your earlier activity (handy when you log out!), press ⌘-K. This gives you essentially the same effect as opening a new Terminal window.

End a Shell Session

When you're finished working on the command line for a while, you *could* simply close the Terminal window, or even quit Terminal, but you *shouldn't*. That would be a bit like turning off your Mac by flipping the switch on the power strip instead of choosing Apple > Shut Down. The proper way to end a shell session in Terminal is to enter `exit`, which gracefully stops any programs you are running in the shell, and then quits the shell program itself. (Starting in El Capitan, the `exit` command also lists cleanup activities it performs, such as saving your latest session history and deleting older sessions. When this list says `[Process completed]`, you know the shell has fully exited. If you use `zsh`, under some circumstances you may have to press Return to see the `[Process completed]` message.)

By default, your Terminal window remains open after you've done this. If you want it to close when you exit, choose Terminal > Settings/Preferences > Profiles > *Current Profile Name* > Shell. (Note: This setting was on the Settings pane in much older versions of macOS.)

From the “When the shell exits” pop-up menu, choose “Close the window.”

Note: For more advanced tips and tricks for working in the zsh shell, see Zsh Tips and Shortcuts, later in this book.

Work with Files and Directories

Much of what you'll need to do on the command line involves working with files in some way—creating, deleting, copying, renaming, and moving them. This chapter covers the essentials of interacting with files and directories.

Create a File

I want to begin by mentioning a curious command called `touch` that serves two interesting functions:

- When supplied with the name of a nonexistent file as an argument, `touch` creates an empty file.
- When supplied with the name of an existing file or folder as an argument, `touch` updates its modification date to the current date and time, marking it as modified.

Try entering the following command:

```
touch file1
```

Now use `ls -l` to list the contents of your current directory. You'll see `file1` in the list. This file that you've just created is completely empty. It doesn't have an extension, or a type, or any contents. It's just a marker, though you could use a text editor, for example, to add to it.

Why would you do this? There are occasionally situations in which a program behaves differently based solely on the existence of a file with a certain name in a certain place. What's in the file doesn't matter—just that it's there. Using `touch` is the quickest way to create such a file.

But for the purposes of this book, the reason to know about `touch` is so you can create files for your own experiments. Since you're creating the files, you can rename, move, copy, and delete them without worrying about causing damage. So try creating a few files right now with `touch`.

> **Note:** Remember, if you want to create a file with a space in the name, put it in quotation marks (`touch "my file"`) or escape the space character (`touch my\ file`).

As for the other use of `touch`—marking a file as modified—you might do this if, for example, the program that saved it failed to update its modification date for some reason and you want to make sure your backup software notices the new version. You use exactly the same syntax, supplying the name of the existing file:

```
touch file1
```

When applied to an existing file, `touch` doesn't affect its contents at all, only its modification date.

Create a Directory

To create a directory (which, of course, appears in the Finder as a folder), use the `mkdir` ("make directory") command. To make a directory called `apples`, you'd enter the following:

```
mkdir apples
```

That's it! A few other potentially useful things to be aware of:

- You can create a new directory in some other location than your current one (for example, you could enter `mkdir ~/Documents/apples`).
- If you want to create a hierarchy of directories—for example, you want to create a directory called oranges inside `~/Documents/fruit/citrus/` and the `fruit` and `citrus` directories don't already exist—add the `-p` flag (for example: `mkdir -p ~/Documents/fruit/citrus/oranges`).

- Spaces, apostrophes, and quotation marks in directory names must be escaped (see Spaces in Paths).

Copy a File or Directory

To duplicate a file (in the same location or another location), use the `cp` ("copy") command. It takes two arguments: the first is the file you want to copy, and the second is the destination for the copy. For example, if you're in your home directory (`~`) and want to make a copy of the file `file1` and put it in the `Documents` directory, you can do it like this:

```
cp file1 Documents
```

The location of the file you're copying, and the location you're copying it to, can be expressed as relative or absolute paths. For instance:

```
cp file1 /Users/Shared
```

```
cp /Users/jk/Documents/file1 /Users/Shared
```

```
cp file1 ..
```

```
cp ../../file1 /Users/Shared
```

If you want to duplicate a file and keep the duplicate in the same directory, enter the name you want the duplicate to have:

```
cp file1 file2
```

Likewise, if you want to copy the file to another location and give the copy a new name, specify the new name in addition to the destination:

```
cp file1 Documents/file2
```

Avoid Overwriting Files When Copying

Look back at the first example above:

```
cp file1 Documents
```

Anything strike you as suspicious about that? We know there's a file called `file1` and a directory called `Documents` in the current directory, so will this command copy `file1` into `Documents` or make a copy in the *current* directory and name the copy `Documents` (potentially overwriting the existing directory)? The answer is: `cp` is smart. The command assumes that if the second argument is the name of an existing directory, you want to copy the file to that directory; otherwise, it copies the file in the current directory, giving it the name of the second argument. It won't overwrite a directory with a file.

But, in fact, `cp` is not *quite* as smart as you might like. Let's say there's *already* a file in `Documents` that's called `file1`. When you enter `cp file1 Documents`, the command happily overwrites the file that's already in `Documents` without any warning! The same goes for duplicating files in the same directory. If the current directory contains files `file1` and `file2`, entering `cp file1 file2` *overwrites* the old `file2` file with a copy of `file1`!

Fortunately, you can turn on an optional warning that appears if you're about to overwrite an existing file, using the `-i` ("interactive") flag. So if you enter `cp -i file1 Documents` and there's already a `file1` in `Documents`, you'll see:

```
overwrite Documents/file1? (y/n [n])
```

Then enter `y` or `n` to allow or disallow the move. "No" is the default.

Because the `-i` flag can keep you out of trouble, I suggest you always use it with the `cp` command. Or, for an easier approach, set up an alias that does this for you automatically; see Create Aliases.

Copy Multiple Files

You can copy more than one file at a time, simply by listing all the files you want to copy, followed by the (single) destination where all the copies will go. For example, to copy files named `file1`, `file2`, and `file3` into `/Users/Shared`, enter this:

```
cp file1 file2 file3 /Users/Shared
```

Copy a Directory

You can use the `cp` command to copy a directory, but you must add the `-r` ("recursive") flag. For instance, given a directory named `apples`, this command would produce an error message:

```
cp apples ~/Documents
```

The correct way to enter the command is as follows:

```
cp -r apples ~/Documents
```

Slashes Away

Avoid putting a slash at the end of the source directory when using `cp -r`. That slash causes the command to copy every item within the directory (but not the directory itself) to the destination. For example, `cp -r apples/ ~/Documents` wouldn't copy the `apples` directory to your `~/Documents` directory, but rather copies the *contents* of the `apples` directory to your `~/Documents` directory—probably not what you want.

If you use tab completion with the `cp` command, be extra careful, because tab completion adds trailing slashes automatically.

Move or Rename a File or Directory

If you want to move a file from one location to another, you use the `mv` ("move") command. This command takes two arguments: the first is what you want to move, and the second is where you want to move it.

For example, if you're in `~` and you want to move `file1` from the current directory to the `Documents` directory, you can do it like this:

```
mv file1 Documents
```

As with `cp`, the location of the file you're moving, and the location you're moving it to, can be relative or absolute paths. Some examples:

```
mv file1 /Users/Shared
```

```
mv /Users/jk/Documents/file1 /Users/Shared
mv file1 ..
```

```
mv ../../file1 /Users/Shared
```

If you want to rename a file, you *also* use the mv (move) command. Weird as it may sound, mv does double duty. When you're renaming a file, the second argument is the new name. For example, to rename the file file1 to file2, leaving it in the same location, enter this:

```
mv file1 file2
```

Tip: Want to move a file from somewhere else to the current directory, without having to figure out and type a long path? You can represent your current location with a period (.), preceded by a space. So, to move file1 from ~/Documents to your current directory, enter mv ~/Documents/file1 . on the command line.

Avoid Overwriting Files When Moving

The mv command works the same way as cp when it comes to overwriting files: it won't overwrite a directory with a file of the same name, but it will happily overwrite files unless you tell it not to do so.

Fortunately, mv supports the same optional -i flag as cp to warn you when you're about to overwrite a file. So if you enter mv -i file1 Documents and there's already a file1 in Documents, you'll see this:

```
overwrite Documents/file1? (y/n [n])
```

You can then enter y or n to allow or disallow the move. Again, "no" is the default.

As with cp, the -i flag is such a good idea that I suggest you get in the habit of using it every single time you enter mv. Alternatively, you can set up an alias that does this for you automatically; see Create Aliases.

Move and Rename in One Go

Since mv can move and rename files, you may be wondering if you can do both operations at the same time. Indeed you can. All it takes is entering the new name after the new location. For instance, if you have a file named file1 and you want to move it into the Documents directory where it will then be called file2, you can do it like this:

```
mv file1 Documents/file2
```

Move Multiple Files

You can move several files at once, by listing all the files you want to move, followed by the (single) destination to which they'll all go. For example, to move files named file1, file2, and file3 into /Users/Shared, enter this:

```
mv file1 file2 file3 /Users/Shared
```

Wildcards with mv

You can use wildcards like * with mv—for example, entering mv *.jpg Pictures moves all the files from the current directory ending in .jpg into the Pictures directory. But when using mv to rename files, wildcards may not work the way you expect. For example, you cannot enter mv *.JPG *.jpeg to rename all files with a .JPG extension to instead end in .jpeg; for that, you must use a shell script (read Change the Extension on All Files in a Folder for an example).

Delete a File

To delete a file, use rm ("remove"), followed by the filename:

```
rm file1
```

Tip: To try this out safely, use touch to create a few files, enter ls to confirm that they're there, then use rm to remove them. Then enter ls again to see that they've disappeared.

You can delete multiple files at once by listing them each separately:

```
rm file1 file2 file3 file4
```

And, of course, you can use wildcards:

```
rm file*
```

Needless to say, you should be extra careful when using the * wildcard with the rm command!

Warning! The rm Command Has No Safety Net

If you put something in the macOS Trash, you can later drag it back out, up until the moment you choose Finder > Empty Trash. But with the rm command (and the rmdir command, described next) there is no such safety net. When you delete files with these commands, they're gone—instantly and completely!

If you want to be especially cautious, you can follow rm with the -i flag, which requires you to confirm (or disallow) each item you're deleting before it disappears forever—for example, rm -i cup* prompts you to confirm the deletion of each file that has a name beginning with cup.

Monterey introduced another option that offers almost as much protection while being less annoying: -I (like -i, but capitalized). With that flag, the rm command requires confirmation only if you're deleting more than three files or if you're recursively removing a directory using the -r flag (discussed just ahead).

Delete a Directory

Just as you can delete a folder in the Finder by dragging it to the Trash, you can delete a directory on the command line with the rmdir ("remove directory") command.

To delete a directory named apples, you can enter this:

```
rmdir apples
```

As with rm, you can delete multiple directories at the same time:

```
rmdir pomegranates pomelos
```

```
rmdir pome*
```

This command works only on empty directories. (A directory can have invisible files created by macOS; don't assume it's empty just because you didn't put anything there.) If you run `rmdir` on a non-empty directory, you get this error message:

```
rmdir: apples: Directory not empty
```

This is a safety feature designed to prevent accidental deletions. If you're sure you want to delete a directory *and* its contents (including subdirectories), use the `rm` command (instead of `rmdir`) with the `-r` (recursive) flag:

```
rm -r apples
```

Use Symbolic Links

If you've been using a Mac for a while, you've probably encountered the concept of an *alias* in the Finder, which is a shortcut to a file or folder stored somewhere else. Aliases are handy if you want quick access to an item in more than one location, but don't want to duplicate it. (Don't confuse a Finder alias with the `alias` command you use to make shortcuts to other commands in Terminal; see Create Aliases.)

Unix, too, has something that acts almost like a Finder alias: a *symbolic link* (or *symlink*). You can create a symbolic link to a file or directory on the command line, and it will (for the most part) behave the way a Finder alias does. For example, if you entered `open alias-name`—in other words, if you used the `open` command on an alias you created in the Finder—the alias's target file or folder would open. (If you're using a relatively old version of macOS, folder aliases may not work on the command line, in which case you'll need a symlink instead.)

Making a symlink is useful when you want to create something that functions on the command line pretty much like an alias in the Finder. You may also find cases where you want to put an app's default folder in another location, but if you replace the original with an alias, it may not work—in most cases, using a symlink instead will do the trick.

To create a symlink, you use `ln` ("link") in this formula:

```
ln -s source destination
```

where *from* and *to* are replaced with the paths to the original item and the symbolic link's new location, respectively.

For example, let's say I want to create a symbolic link to my `~/Pictures` directory and put it on my desktop. I'd do it like this:

```
ln -s ~/Pictures ~/Desktop
```

The key thing to remember is that the *source* argument is the path to the item you want to link to, *including its filename,* and the *destination* argument is the path to where you want the symlink to be stored (with or without a filename). If you leave off the filename (as in the example above), the symlink will have the same name as the original file or directory. However, if you want the symlink to have a different name, you can specify that in the *to* argument, like this:

```
ln -s ~/Pictures ~/Desktop/photographs
```

If you create a symlink in Terminal and look at the resulting icon in the Finder, you'll see a little arrow in the lower left, just like an alias.

Work with Programs

Every command that you use on the command line, including merely listing files, involves running a program. (So, in fact, you've been using programs throughout this book!) However, some aspects of using programs on the command line aren't entirely obvious or straightforward. In this chapter, I explain some of the different types of programs you may encounter and how to run them (and stop them).

I also show you how to edit files on the command line, and I talk about *shell scripts,* a special kind of program you can create to automate a sequence of tasks.

Learn Command-Line Program Basics

If you've been reading this book in order, you already know many basics of running programs on the command line. Each time you enter a command such as `ls` or `cp` or `pwd`, you're running a program—and we saw how to change program options and supply additional parameters with arguments and flags earlier (in What Are Commands, Arguments, and Flags?). However, I think you should know a few other important facts about running programs.

Command-line programs come in a few varieties, which I'll lump together in three broad categories. (These are my own terms, by the way; other people may categorize them differently.) You'll have an easier time using the command line if you're aware of the differences.

Basic Programs

Most command-line programs you use simply do their thing and then quit automatically. Enter `ls`, for instance, and you instantly get a list of files, after which point `ls` is no longer running. Some of these single-shot programs produce visible output (`date`, `ls`, `pwd`, etc.); some normally provide no feedback at all unless they encounter an error (`cp`, `mv`, `rm`, etc.). But the point is: they run only as long as is needed to complete their task, without requiring any interaction with you other than the original command (with any flags and arguments).

Interactive Programs

A second type of program asks you for an ongoing series of new commands, and in some cases doesn't quit until you tell it to. For example, the command-line program used to change your password is `passwd`. If you enter `passwd`, you see something like the following:

```
Changing password for jk.

Old password:█
```

You type your old password and press Return, and then the program gives you another prompt:

```
New password:█
```

Type in a new password and you get yet another prompt:

```
Retype new password:█
```

Reenter your new password, as long as it matches the first one, the program changes your password and exits without any further output.

Note: This procedure changes the password for your Mac's user account, which applies everywhere (not just on the command line).

Programs of this sort include `ssh`, which lets you Log In to Another Computer, and `sftp`, which lets you transfer files between computers, among many others. If you're running an interactive program, want to quit it, and can't find an obvious way to do so, try pressing Control-C (see Stop a Program for more possibilities).

Note: Starting in 10.11 El Capitan, whenever you use a command that prompts you for a password, such as `passwd` or `sudo`, a key 🔑 icon appears after the password prompt to remind you that whatever you type will not be shown on screen. It also means you're in Secure Input mode, so third-party apps (such as 1Password, Keyboard Maestro, and TextExpander) can't enter text on your behalf.

Full-Window Programs

The third broad category of programs is full-window programs—those that are interactive but, instead of handling input and output on a line-by-line basis, take over the entire Terminal window (or tab). You've already tried a few of these—`less` and `man` are examples. Some full-window programs helpfully display hints at the top or bottom of the window showing what commands you can use; others require that you've memorized them (or can look them up in a `man` page, perhaps in another window). As with other interactive programs, pressing Control-C usually lets you exit a full-window program if you can't find another way to do so.

Change Your Terminal Type

A curious feature of full-window programs such as `less`, `top`, and `man` is that once you quit them, everything they previously displayed on screen disappears; for example, you can't scroll back to see something from a `man` page once you quit `man`.

This behavior (among others) is determined not by your shell but by the specific kind of terminal that Terminal happens to be emulating at any given time. By default, that terminal type is something called xterm-color. Without getting into any tedious details, let's just say that xterm-color has many virtues, but some people dislike the way it handles full-window programs. If you're one of those people, you can easily switch to a different terminal type.

Follow these steps:

1. Choose Terminal > Settings/Preferences > Profiles > *Profile Name* > Advanced.
2. Choose vt102 from the "Declare terminal as" pop-up menu.
3. Close the Settings/Preferences window.

The change takes effect beginning with the next shell session you open in Terminal.

Run a Program or Script

Often, running a program requires nothing more than typing its name and pressing Return. However, the process can be a bit trickier in certain cases. Read on to discover how to run programs located in unusual places, as well as scripts (programs packaged in a somewhat different form).

How Your PATH Works

As you know already (see Understand How Paths Work), each file on your Mac has a path—a location within the hierarchy of directories. So a path of `/Users/jk/Documents/file1` means `file1` is in the `Documents` directory, which is in turn in the `jk` directory, which is in `Users`, which is at the top, or root, level of the disk (signified by the initial `/`).

But there's another, specialized use of the term *PATH*: when capitalized like this, it refers to a special variable your shell uses that contains a list of all the default locations in which a shell can look for programs.

To run a program, your shell must be able to find it. But so far, all the commands you've entered have been "bare" program names without specific paths given. For example, to run `less`, you simply enter `less`, but in reality the program you're running is stored in `/usr/bin`. Looking everywhere on your disk for a program would be time-consuming, so how can your shell find it in order to run it? The answer is that when you enter a command without an explicit path, the shell automatically looks in several predetermined locations. That list of locations, which happens to include `/usr/bin`, is your PATH.

By default, your PATH includes all of the following directories:

```
/bin
/sbin
/usr/bin
/usr/local/bin
/usr/sbin
```

A program in any of these locations is said to be "in your PATH." You can run a program in your PATH, regardless of your current location in the directory structure, simply by entering its name. I encourage you to look through these directories (try `ls -l /bin`, `ls -l /sbin`, and so on) to get an idea of the hundreds of built-in programs and where they're located.

> **Tip:** To see the current contents of your PATH, enter `echo $PATH`. Each valid directory is separated from the next by a colon—for example: `/usr/local/bin:/usr/bin:/bin:/usr/sbin:/sbin`.

Most programs you'll need to run are already in your PATH, and if you download or create new programs, you can put them in one of these locations to make sure you can run them just by entering their names. But what about programs that aren't in your PATH? You can either enter the program's full or relative path (for example, `/usr/local/bin/stuff` or `../software/myprogram`), or you can expand your PATH to include other directories (I explain how in Modify Your PATH).

Run a Program

To summarize, you can run a program in any of three ways, depending on where the program is located, your current position in the directory structure, and what's in your PATH:

- **By relative or absolute path:** You can *always* run a program by entering its complete path, such as `/usr/bin/less`, or its relative path from the current location, for example `apples/oranges/program`.
- **In the current directory:** If you're in the same directory as the program you want to run, you might think you could just enter the program's name, but that doesn't work. Instead, you must precede the name with `./` (and no space). For example, to run a program named `counter` in the current directory, enter `./counter`.
- **In your PATH:** To run a program anywhere in your PATH, simply enter the program's name—for example, `less`, `mkdir`, or `man`.

Running Multiple Programs on One Line

Ordinarily, if you want to run two commands in a sequence, you enter the first command, let it run, and then enter the second one on a new line. However, sometimes it’s more convenient to tell the command line: “Hey, just run these commands one after the other instead of making me wait to enter the next one.” To do this, type the first command, then a semicolon (`;`), an optional space, and the second command. (You can chain more than two commands together in this way.) Not every command can work as part of a chain, but most do.

To learn about another way of running multiple programs on one line, in which one command’s output supplies the input for the next command, see Pipe and Redirect Data.

Which Programs Can I Run?

In the Finder, you have but to glance in the Applications folder and its subfolders to see most of the GUI apps you can run. But on the command line, which programs are available to you? Since they’re stored in so many places, it’s not obvious—and the programs mentioned in this book are a tiny subset of what comes with your Mac.

Well, there’s a trick. It’s weird, but here goes. While in the `bash` shell (this doesn’t work with `zsh`), press either Tab or Esc twice. (Hint: with Esc, you have to have a *very* brief pause between the two presses. If the presses are too fast or too slow, it won’t work.) You’ll see a prompt, such as:

```
Display all 1702 possibilities? (y or n)
```

Press `y`, and your window fills up in a paged (`less`-style) display of every command-line program on your Mac. To find out what one of these programs does, you can right-click (or Control-click) it and choose “Open man Page” from the contextual menu.

Run a Script

In macOS, as in other varieties of Unix, the programs you run are usually compiled *binary* files. If you were to open them in a text editor, they’d look like nothing but garbage characters, because they’ve been put into an optimized, self-contained, machine-friendly format for maximum performance. However, another broad category of programs consists of human-readable text that's interpreted by the computer as it runs instead of being compiled in advance. Programs in this category are often referred to as *scripts,* and they're often used to automate or simplify repetitive activities. Just as AppleScript provides a way of writing human-readable programs that run in the Mac's graphical environment, scripts of various kinds can run from the command line.

A *shell script* is a series of instructions interpreted, or run, by the shell itself. So, a shell script could consist of little more than a list of commands, just as you would type them manually in a Terminal window. Run the script, and the shell executes all those commands one after the other. (In fact, shell scripts can use variables, conditional tests, loops, math, and much more—I introduce you to these items later, in Add Logic to Shell Scripts.) I explain the basics of creating a simple script ahead in Create Your Own Shell Script. By convention, shell scripts usually have an extension of `.sh` (regardless of which shell they use); see Run a Shell Script from the Finder for an exception.

Other kinds of scripts are written in scripting languages such as Perl, Python, and Ruby, and run by the corresponding interpreter. Perl scripts, by convention, end in the `.pl` extension, PHP scripts in `.php`, Python scripts in `.py`, and Ruby scripts in `.rb`. In recent versions of macOS, you can even use Swift as a scripting language, with filenames ending in `.swift`.

Note: Refer back to Scripting Languages Were Deprecated for information on the future of these languages in macOS.

Regardless of a script's extension, it's considered good programming practice to include the name and location of the interpreter that should process it on the first line of the script. For example, if a shell script is intended to be interpreted by the `sh` shell, the first line should be:

```
#!/bin/sh
```

The `#!` at the beginning of this line, called a "shebang," is a marker indicating that what follows it is the path to the interpreter. (You can examine a script using, say, `less` or `cat` to see if it has such a line.)

Because the interpreter is spelled out right in the script, you can run the script just as you would any other program, by entering its name (if it's in your PATH) or its path, and the shell you're currently using need not be the same as the shell specified in the script.

> **Tip:** All the built-in shells (`bash`, `csh`, `dash`, `ksh`, `sh`, `tcsh`, and `zsh`) are located in `/bin`. Interpreters for included scripting languages are located in `/usr/bin` by default, but if you install new versions yourself, they may be located elsewhere.

However, if a script doesn't include that line, you must tell it explicitly which shell or other interpreter to run it with. You do that by entering the interpreter's name with the path to the script as an argument. For example:

```
sh ~/Documents/my-shell-script.sh

perl ~/Documents/my-perl-script.pl

php ~/Documents/my-php-script.php

python ~/Documents/my-python-script.py

ruby ~/Documents/my-ruby-script.rb

swift ~/Documents/my-swift-script.swift
```

> **Note:** Recall that PHP was removed from macOS in version 12.0 and Python in version 12.3, but you can install them manually if needed. If you're using Apple-supplied versions of any of the above except Swift in Catalina or later, you will see a notice when you run a script that the scripting language has been deprecated (see Scripting Languages Were Deprecated), but as long as the language is installed, the script should function normally.

If you've created automations using the Shortcuts app in Monterey or later, you can also run shortcuts from the command line. Jason Snell offers instructions in his article Run shortcuts from the Mac command line. (To learn more about Shortcuts, read Rosemary Orchard's book *Take Control of Shortcuts*.)

Running Shell Scripts Outside the Shell
You don't have to be in the Terminal app to run a shell script! You can also run shell scripts from within numerous other apps and environments, including:

- AppleScripts
- Automator workflows
- Keyboard Maestro macros
- TextExpander snippets

I cover these and numerous other examples in my book *Take Control of Automating Your Mac*.

Run a Program in the Background

Most of the time when you run a program, it does its thing, and then you quit it (or it quits by itself). While it is running—whether that takes a second or an hour—it takes over your shell and thus the Terminal window or tab in which the shell is running. If you expect a program to take some time to complete its task, or if you want it to keep running even after you exit the shell, you can run it in the background. Background programs let you do other tasks in the same Terminal window or tab, and, if necessary, they keep going even after you quit Terminal.

To run a program in the background, you simply type the program name (and any flags or arguments) followed by an optional space and an ampersand (`&`). For example, suppose you want to compress a folder containing hundreds of large files. Ordinarily, you might use a command like `zip -r archive.zip apples`. To run that command in the background instead, enter this:

```
zip -r archive.zip apples &
```

While a program is running in the background, you'll see no feedback or output. If it's a program that simply accomplishes a task (such as copying or compressing files) and then quits automatically, then you'll see a message stating that it's finished—not *immediately* afterward, but the next time you execute a command or even just press Return to create a new command line. The message saying a process is finished looks something like this:

```
[1]+  Done          zip -r archive.zip apples
```

Note: Programs designed to run in the background *at all times* are called daemons (pronounced "demons"). Examples include database and web servers, firewalls, and some backup programs. You wouldn't use the term "daemon," however, for an ordinary program you opt to run in the background temporarily.

See What Programs Are Running

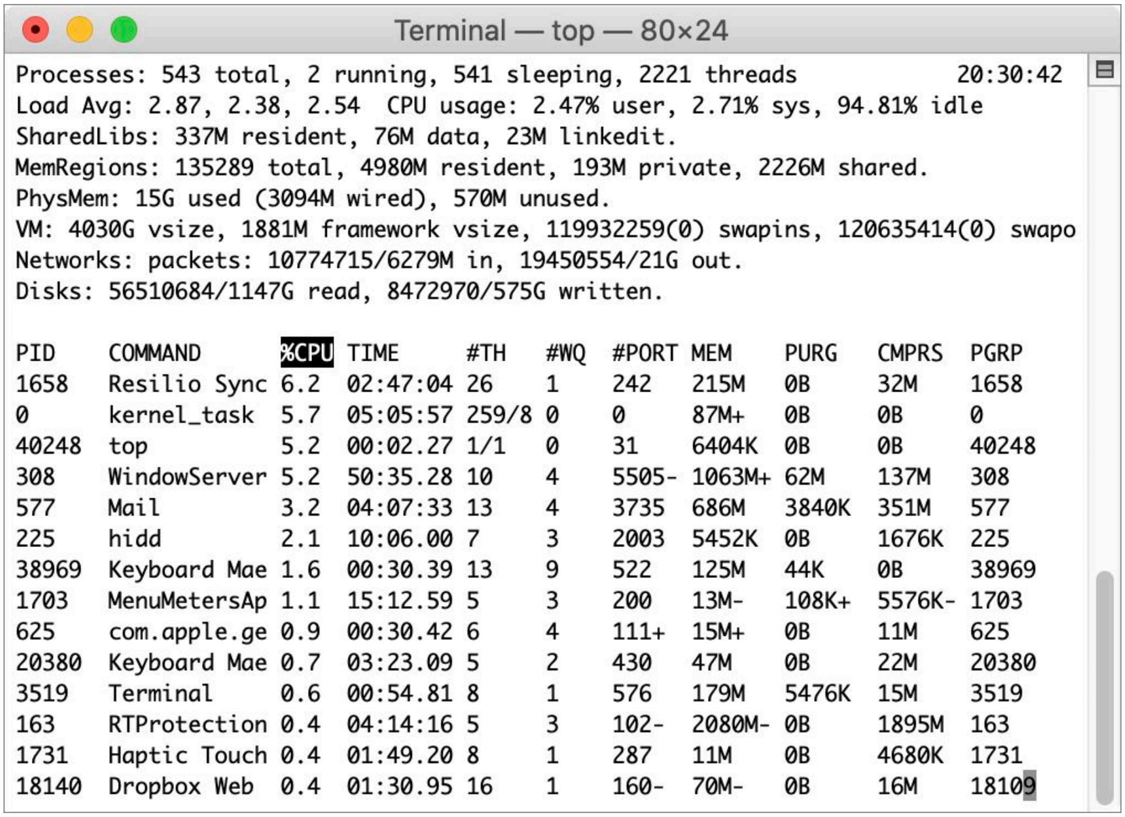
Here's a thought question: How many programs are running on your Mac right now? If you glance at the active icons in your Dock and conclude that the number is, say, a dozen, you haven't even scratched the surface. For example, as I type these words, my Dock tells me I have 16 programs running, but in reality the total is over 500! Besides the visible programs like Mail and Safari, that figure includes background programs that are part of macOS—the Spotlight indexer, Time Machine, logging tools, and many others that perform important but little-noticed functions behind the scenes. It also includes my `zsh` shell running in Terminal, and every program running in that shell.

Note: Roughly speaking, the term "process" is used to describe programs (of any sort) that are actively running, as opposed to those that are merely occupying space on your disk. The commands and procedures I describe in this section are concerned only with active programs, and therefore I use the term "process" to describe them.

You may be aware of Activity Monitor (in `/Applications/Utilities`), which lists all this information and more. In the command-line environment, too, you can list all your Mac's processes (visible and invisible) and get a variety of useful facts about them. The two most commonly used command-line programs for discovering what's running on your Mac are `top` and `ps`.

Top

The top command is the nearest command-line equivalent to Activity Monitor. Enter top and you get a full-window list of all your running processes, updated dynamically. **Figure 5** shows an example.



```
Processes: 543 total, 2 running, 541 sleeping, 2221 threads            20:30:42
Load Avg: 2.87, 2.38, 2.54  CPU usage: 2.47% user, 2.71% sys, 94.81% idle
SharedLibs: 337M resident, 76M data, 23M linkedit.
MemRegions: 135289 total, 4980M resident, 193M private, 2226M shared.
PhysMem: 15G used (3094M wired), 570M unused.
VM: 4030G vsize, 1881M framework vsize, 119932259(0) swapins, 120635414(0) swapo
Networks: packets: 10774715/6279M in, 19450554/21G out.
Disks: 56510684/1147G read, 8472970/575G written.

PID    COMMAND      %CPU TIME     #TH   #WQ  #PORT MEM    PURG   CMPRS  PGRP
1658   Resilio Sync 6.2  02:47:04 26    1    242   215M   0B     32M    1658
0      kernel_task  5.7  05:05:57 259/8 0    0     87M+   0B     0B     0
40248  top          5.2  00:02.27 1/1   0    31    6404K  0B     0B     40248
308    WindowServer 5.2  50:35.28 10    4    5505- 1063M+ 62M    137M   308
577    Mail         3.2  04:07:33 13    4    3735  686M   3840K  351M   577
225    hidd         2.1  10:06.00 7     3    2003  5452K  0B     1676K  225
38969  Keyboard Mae 1.6  00:30.39 13    9    522   125M   44K    0B     38969
1703   MenuMetersAp 1.1  15:12.59 5     3    200   13M-   108K+  5576K- 1703
625    com.apple.ge 0.9  00:30.42 6     4    111+  15M+   0B     11M    625
20380  Keyboard Mae 0.7  03:23.09 5     2    430   47M    0B     22M    20380
3519   Terminal     0.6  00:54.81 8     1    576   179M   5476K  15M    3519
163    RTProtection 0.4  04:14:16 5     3    102-  2080M- 0B     1895M  163
1731   Haptic Touch 0.4  01:49.20 8     1    287   11M    0B     4680K  1731
18140  Dropbox Web  0.4  01:30.95 16    1    160-  70M-   0B     16M    18109
```

Figure 5: In the top window, you get a list of all the processes currently running on your Mac.

By default, the top command lists several pieces of information for each process, including the following particularly interesting ones: PID (process ID), COMMAND (the process name), %CPU (how much CPU power the process is using), TIME (how long the process has been running), and MEM (how much RAM the process is using).

I won't go into great detail about everything you see here (try man top to learn more), but I do want to call your attention to a few salient points and offer some top tips:

- **Pruning the list:** You almost certainly have many more processes than can fit in your window at one time, even if you make your window very large. So you can restrict the number of items top shows at a time using the -n (number) flag, followed by the number of items to show (top -n).

- **Sorting the list:** By default in recent versions of macOS, top lists processes in reverse order of %CPU, which means the processes at the top of the list are the ones that have been using the most CPU power recently. You can adjust the sort order with the -o (order) flag—for example, enter top -o pid to list processes in order of process ID (roughly, in order of how recently they were launched), or enter top -o rsize to list processes in order of RAM usage.
- **Top at the top:** Depending on what else is running on your Mac at the moment, top itself may be at or near the top of the list, even when sorted by CPU usage. Don't be alarmed: the effect is caused by the way top gathers its data.
- **Customizing the list:** You can combine flags to customize your display. For example, enter top -n 20 -o cpu to list only the top 20 processes by CPU usage.
- **Quitting:** To quit top, just type the Q key (by itself).

Ps

Whereas top is dynamic, you may want simply to get a static snapshot of the processes running at any moment. For that, the best command is ps ("process status"). If you enter ps by itself, you get a list of your processes running in terminals—which usually means the Terminal app. In all likelihood, this is just zsh or bash itself.

The list includes the PID, the TTY (or terminal name), time since launch, and command name for each process:

```
  PID TTY           TIME CMD

22635 ttys001    0:00.06 zsh
```

You can expand the amount of information that ps provides using flags. For example, to include not only processes in the current shell session but also those from other sessions (yours or other users'), enter ps -a. To show processes that aren't running in a shell at all (including regular macOS apps and background processes), enter `ps -x`. Combine the two (`ps -ax`) to show all the processes running on your Mac.

Of course, although `ps -ax` provides lots of information, it might be too much to be useful. You can filter the output from the `ps` command by using a couple of spiffy Unix tricks. First, add the pipe (`|`) character (type Shift-\) to channel the output from `ps` into another program. (For more on the pipe, see Pipe and Redirect Data, later.) The other program, in this example, is `grep`, a powerful pattern-matching tool we'll see again in Get a Grip on grep. So, enter `ps -ax | grep` followed by a space and some text, and what you get is a list of all and only the running processes whose listing includes that text. For example, to list all processes running from inside your `/Applications` directory, enter:

```
ps -ax | grep /Applications
```

Note: A curiosity of this command is that the `grep` process itself appears in the list, because `grep` includes `/Applications` as an argument! If that bothers you and you want to exclude `grep` itself, add the following after `/Applications` and a space: `| grep -v grep`. The same applies for the next example.

Or, to show only the processes whose names include the characters `sys` (in any combination of upper- and lowercase), try this:

```
ps -ax | grep -i sys
```

Stop a Program

As we've seen, most command-line programs quit automatically when they finish performing their functions, and full-window programs usually have a fairly obvious way of quitting them (for example, pressing `Q` in the case of `less` or `man`. However, if a program doesn't quit on its own, or if you need to unstick one that's stuck (even if it's a graphical macOS app!), you can use one of several techniques.

Ask Politely

If a command-line program won't quit on its own, the first thing to try is pressing Control-C. In this context, it's approximately like pressing ⌘-Q in a regular macOS app—it tells the process to quit, but to do so in a controlled way (saving open files and performing any other necessary cleanup operations).

Kill (Humanely)

What if you want to stop a program that's not running in the current shell? If it's a graphical macOS app, or an invisible background process, or a program running in another shell, you can send it a "Quit" signal remotely. The command you use to do this is `kill`. That sounds extreme, but, in fact, when `kill` is used on its own, it sends a program the same sort of polite request to terminate that Control-C does.

> **Note:** You can kill only processes you own (that is, ones started under your user account). To kill another user's processes, you must use `sudo` (see Perform Actions as the Root User).

The catch is that you have to know how to refer to the program you want to kill. Here there are two options:

- **By PID:** If you can find the process's PID (process ID)—using `top`, `ps`, or even Activity Monitor—you can simply enter `kill` followed by that number. For example: `kill 1342`
- **By name:** If you don't know the process's PID, or can't be bothered to find out—but do know its name—you can quit it by name using a variant of `kill` called `killall`. Simply follow `killall` with the program's name. For example: `killall Safari`

  You must enter the name exactly as it appears in `top`, `ps`, or Activity Monitor. For example, if you want to quit Excel, you must enter `killall "Microsoft Excel"` (quotation marks added because there's a space in the name).

Kill (with Extreme Prejudice)

If a program fails to respond to Control-C or to the standard kill or killall command, it's time to pull out the big guns. By adding the -9 flag to the kill command, you turn a polite request into a brutal clobbering that can terminate almost any process.

When you use the kill -9 command, you must give it the process's PID; the -9 flag doesn't work with killall to force-quit a process by name. For example:

```
kill -9 1342
```

If even kill -9 doesn't stop a process, and I've seen that happen more than once, it is likely stuck beyond the power of any software command, and if logging out of your user account doesn't solve the problem, your only choice is to restart the computer.

Edit a Text File

Earlier I showed you how to view the contents of text files, but you may also need to modify them. For that, you can work with any of several command-line text editors. Using a command-line text editor is often quicker and easier than opening a text file in a program like TextEdit—especially for files that don't appear in the Finder—and is less likely to cause problems with file formats or permissions.

If you ask a hardcore Unix geek what text editor they use, they will probably answer vi. (That's "vee-eye," not "vie," by the way.) It's a very powerful text editor that's been around forever, and because a lot of programmers cut their teeth on vi and then proselytized future generations, it's become a sort of badge of honor to be skilled in using vi.

macOS includes vi, but I'm not going to tell you how to use it. As command-line programs go, vi has the most opaque user interface I've seen. Until you get used to vi's oddities and memorize its commands, you can't even change a letter in a text document without referring to a manual. Powerful or not, from a usability standpoint, vi is hideous.

I just want you to know about vi so that when someone asks you why you don't use it, you can give the correct response: "Life is too short."

Happily, you can use several other fine text editors. There's the venerable emacs, which is less obnoxious than vi while still being fabulously flexible. But I'm going to recommend what you might think of as the TextEdit of command-line text editors: a simple, straightforward, and adequately powerful program called nano.

From pico to nano and Back

The nano editor is an "enhanced clone" of an earlier editor called pico; they have almost identical interfaces and feature sets. Years ago, nearly all Unix and Unix-like operating systems replaced pico with nano, but put in a link so that you could still type pico, but the (nearly indistinguishable) nano would open. That's how it has been on my Mac since I can remember.

However, somewhat bizarrely, in macOS Monterey 12.3, Apple removed nano and reverted back to pico, putting in a link for those who had become accustomed to using nano. So now, if you type nano, it's actually pico that runs, not the other way around. (Scuttlebutt on the internet suggests Apple did this for licensing reasons.)

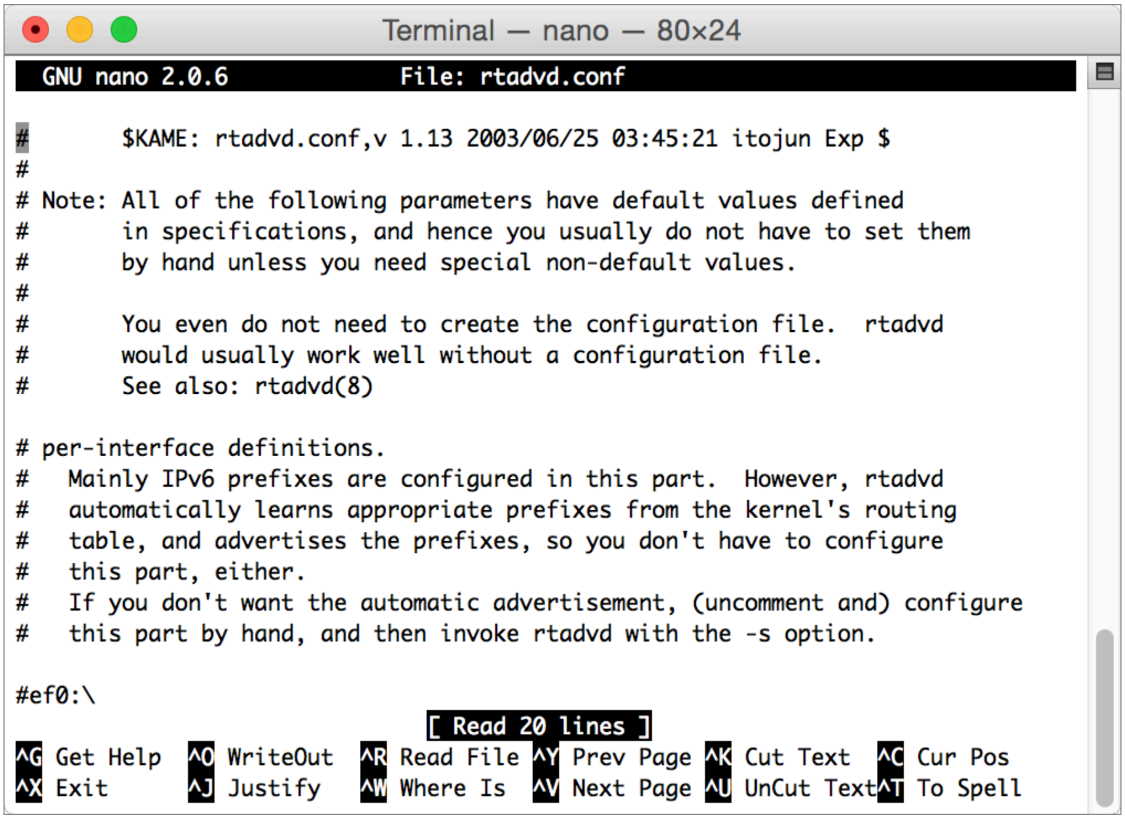
For all practical purposes, none of this makes any difference. Everything in this book works identically whether you type nano or pico. Even though pico is what really runs, behind the scenes, I'm leaving nano in the text for historical reasons (and because it's what you'll most likely encounter in other operating systems). But if you want to type pico whenever you see nano, that's fine.

If, for some reason, it *really matters* to you that you use nano and not pico, you can install nano yourself. First install Homebrew, then type brew install nano and Bob's your uncle.

To edit a text file in nano, use a command like the following:

```
nano file1
```

If file1 is already present, nano opens it; otherwise, it opens a blank file that will be called file1. **Figure 6** shows a text file open in nano.



```
  GNU nano 2.0.6                File: rtadvd.conf

#       $KAME: rtadvd.conf,v 1.13 2003/06/25 03:45:21 itojun Exp $
#
# Note: All of the following parameters have default values defined
#       in specifications, and hence you usually do not have to set them
#       by hand unless you need special non-default values.
#
#       You even do not need to create the configuration file.  rtadvd
#       would usually work well without a configuration file.
#       See also: rtadvd(8)

# per-interface definitions.
#   Mainly IPv6 prefixes are configured in this part.  However, rtadvd
#   automatically learns appropriate prefixes from the kernel's routing
#   table, and advertises the prefixes, so you don't have to configure
#   this part, either.
#   If you don't want the automatic advertisement, (uncomment and) configure
#   this part by hand, and then invoke rtadvd with the -s option.

#ef0:\
                             [ Read 20 lines ]
^G Get Help  ^O WriteOut  ^R Read File ^Y Prev Page ^K Cut Text  ^C Cur Pos
^X Exit      ^J Justify   ^W Where Is  ^V Next Page ^U UnCut Text^T To Spell
```

Figure 6: A text file open in the nano text editor. The menu of keyboard controls is at the bottom of the window.

Note: You can select text in a nano screen using your mouse, you can copy it using Edit > Copy or ⌘-C, and you can even Option-click to move the cursor to an exact spot in the text. See the sidebar Using a Mouse in Terminal for more tips. But in general, you should ignore your mouse in nano.

One of the reasons nano is easy to use is that editing is straightforward. To insert text at the cursor location, simply type—or paste the contents of your clipboard by choosing Edit > Paste or pressing ⌘-V. To delete the character to the left of the cursor, press the Delete key; to delete the character under the cursor, press the Forward Delete key (if your keyboard has one) or Fn-Delete. To delete the entire current line, press Control-K.

Tip: The nano editor doesn't have an Undo command as such, but if you cut a line of text with Control-K and want to restore it, you can press Control-U to "uncut" it.

Other than those basics, here are the most important things you should know how to do in `nano`:

- **Save:** To save the current file, press Control-O (WriteOut).
- **Quit:** To quit `nano`, press Control-X (Exit). If you've made any changes to the document that you haven't yet saved, `nano` prompts you to save the file before exiting. Press `N` to discard changes and quit immediately, `C` to cancel and stay in `nano`, or `Y` to save changes and exit. If you do save changes, `nano` verifies that you want to keep the existing filename (if you don't, you can type a new one). Press Return after verifying the filename.
- **Find:** To find text within the file, press Control-W (Where Is). Type the text you're searching for (case doesn't matter) and press Return. The cursor jumps to the next spot in the document where that string appears. Repeat this procedure to do additional searches.

Those commands alone should enable you to do almost everything you need to do in `nano`. To learn about additional `nano` features and shortcuts, press Control-G to view its online help.

Create Your Own Shell Script

Before I wrap up this discussion of running programs, I want to give you a tiny taste of creating your own shell scripts. Scripting is a bit like learning chess: you can pick up the basics in a few minutes, but it may take years to master all the subtleties. So I'm not going to teach you anything about *programming* as such, just the mechanics of creating and using a simple script. I want you to have enough familiarity with the process that you can successfully reproduce and run shell scripts you may run across in magazines, on websites, or even in this book (see Command-Line Recipes, which includes a couple of shell scripts). Later on, for those who are interested in learning a bit more, I've included instructions on how to Add Logic to Shell Scripts.

You can create and run a shell script in six easy steps; in fact, you can arguably combine the first four into a single process. But one way or another, you must make sure you've done everything in the list ahead.

Step 1: Start with an Empty Text File

Scripts are plain text files, so you should begin by creating one in a text editor. You can make a shell script in TextEdit, BBEdit, or even Word, but that requires extra steps. So I suggest using `nano`, as described in Edit a Text File. For the purpose of demonstration, name your script `test.sh`. (Remember from Run a Script that the `.sh` extension isn't mandatory, but it can help you keep track of which files are shell scripts.)

Before you create this file, I suggest using `cd` (all by itself!) to ensure that you're in your home directory. (You can put scripts anywhere you want, but for now, this is a convenient location.) That done, enter `nano test.sh`. (As a reminder, the extension doesn't really matter—and it certainly doesn't depend on what shell you're using. It's just an arbitrary convention to end shell scripts in `.sh`.) The `nano` text editor opens with a blank file.

Step 2: Insert the Shebang

The first line of your script should include the "shebang" (`#!`) special pointer (see Run a Script) to the shell it will use. Since we're focusing mainly on the `zsh` shell, we'll use that one. Type the following line:

```
#!/bin/zsh
```

> **Note:** Every script in this book works exactly the same way with `bash` as the interpreter. So, if you prefer, feel free to use `#!/bin/bash` instead. Just be aware that the two are not 100% interchangeable—some scripts you encounter might require one or the other.

Step 3: Add One or More Commands

Below the shebang line, you enter the commands your script will run, in the order you want them executed. Your script can be anything from a single one-word command to thousands of lines of complex logic.

For now, let's keep things simple. Starting on the next line, type this:

```
echo "Hello! The current date and time is:"
date
echo "And the current directory is:"
pwd
```

The `echo` command simply puts text on the screen—and you've seen the `date` and `pwd` commands. So, this script displays four lines of text, two of which are static (the `echo` lines) and two of which are variable.

Step 4: Close and Save the File

To save the file, press Control-O and press Return to confirm the filename. Then press Control-X to exit `nano`.

Step 5: Enable Execute Permission

The only slightly tricky thing about running scripts—and the step people forget most often—is adding execute (run) permission to the file. (I say more about this later, in Understand Permission Basics.)

To do this, enter:

```
chmod u+x test.sh
```

Step 6: Run the Script

That's it! To run the script, enter `./test.sh`. It should display something like this:

```
Hello! The current date and time is:
Wed Feb 5 19:58:21 PST 2020
And the current directory is:
/Users/jk
```

For fun, try switching to a different directory (say, `/Library/Preferences`) and then run the script again by entering `~/test.sh`. You'll see that it shows your new location.

Any time you need to put a new script on your system, follow these same steps. You may want to store the scripts you create somewhere in your PATH (see How Your PATH Works), or add to your PATH (see Modify Your PATH), to make them easier to run.

Shell scripts can be much more than simple lists of commands. If you want to explore more advanced capabilities, skip ahead to Add Logic to Shell Scripts.

Customize Your Defaults

Now that you know the basics of the command line and Terminal, you may find some activities are a bit more complicated than they should be, or feel that you'd like to personalize the way your shell works to suit your needs. One way to exercise more control over the command-line environment is to customize your defaults, which are stored in a special file your shell reads every time it runs. In this chapter, I explain how this process works and how you can use it to save typing, customize your prompt, and more.

About Startup Files

A *startup file* (also known as an initialization file, or informally as a profile) is a file your shell reads every time you start a new session that can contain a variety of preferences for how you want the shell to look and behave. Among other things, this file can customize your PATH variable (see How Your PATH Works), add shortcuts to commands you want to run in a special way, and include instructions for personalizing your prompt. I cover just a few basics here.

For complicated reasons, `zsh` and `bash` load different startup files under different circumstances—and in some cases, more than one such file loads when you begin a session. Rather than detail which files do what and when, I'll cut to the chase: for most people, the optimal startup file to use for your defaults under `zsh` is `~/.zshrc`, and for `bash`, the optimal file is `~/.bash_profile.` Those are the ones I cover in this book.

Fortunately, most of the common configurations are the same for both `zsh` and `bash` (I point out some minor differences ahead). So, if you're moving from `bash` to `zsh` and you already customized `.bash_profile` but don't yet have a `.zshrc` file, you can make a great first pass by duplicating the old file but with a new name:

```
cp ~/.bash_profile ~/.zshrc
```

> **Note:** Customizations you make in `.zshrc` or `.bash_profile` apply only in a shell session; they aren't used by shell scripts (see Create Your Own Shell Script). As a result, when writing a script, you should always spell out complete paths and assume default values for all variables.

Edit .zshrc or .bash_profile

To edit `.zshrc` or `.bash_profile` in `nano`, simply enter the following:

`nano ~/.zshrc` (for `zsh`) or

`nano ~/.bash_profile` (for `bash`)

If the file already exists, `nano` opens it for editing; if not, it prompts you to create the file when you save or quit the program.

This file is a simple text file, and unlike shell scripts, it doesn't use a shebang. Just add one or more lines to specify the changes that you want (as described on the following pages). When you're finished editing `.bash_profile`, save it (Control-O) and close it (Control-X). Ordinarily, the changes take effect with the next shell session (window or tab) you open. To load the modified profile immediately, enter `source .zshrc` (for `zsh`) or `source .bash_profile` (for `bash`).

Create Aliases

In the Finder, an alias is a small file that serves as a pointer to another file (for something comparable to Finder aliases on the command line, refer to Use Symbolic Links). In the command-line environment, however, the word *alias* means a shortcut in which one command substitutes for another.

For example, suppose you're used to command-line conventions from DOS and Windows, in which you enter `dir` (directory) to list your files. If you want to use that same command in macOS, you can make an alias, so that entering `dir` runs the `ls` command. Or, maybe there's a lengthy command you use frequently, and you want to run it with fewer keystrokes. No problem: you can use (for instance) `pp` to mean `cp *.jpg ~/Pictures/MyPhotos`.

To create an alias, put a new line in `.zshrc` or `.bash_profile` consisting of the word `alias`, a space, the shortcut you want to use, and `=""` with the command you want to run inside the quotation marks. For example, to use the command `dt` as a shortcut for the `date` command, enter this:

```
alias dt="date"
```

Warning! Be sure *not* to include spaces on either side of the `=` sign in configuration files. Unpredictable and undesirable results may occur.

Aliases can include flags and arguments, and if you enter a shortcut that's identical to an existing command, your alias takes precedence. For example, if you always want to show file listings in the long format, instead of typing `ls -l` every time, you can create an alias so typing `ls` gives you the same result:

```
alias ls="ls -l"
```

Or, suppose you've taken my advice to heart to always use the `-i` flag with `cp` (copy) and `mv` (move), to display a warning if the command is about to overwrite an existing file. You could add aliases to new, easy-to-remember commands like `copy` and `move`, respectively, with those options pre-configured:

```
alias copy="cp -i"
```

```
alias move="mv -i"
```

Warning! You could set up aliases such that entering `cp` or `mv` would include the `-i` flag, but I recommend against it because you might get into a habit of using `cp` and `mv` carelessly, assuming you'll be warned of any impending overwrite. That could lead to data loss if you find yourself using the command line on a computer that doesn't have the same aliases configured.

Modify Your PATH

As I explained in How Your PATH Works, when you run a program by entering just its name, your shell looks for it in several predetermined directories. You may want to specify additional places where programs or scripts are located, to make it easier to run them. For example, if you're experimenting with your own scripts and you store them all in `~/Documents/scripting`, you should add that directory to your PATH.

To add a directory to your PATH, put this in `.zshrc` or `.bash_profile`:

```
export PATH=$PATH:/your/path/here
```

For example, to add the directory `~/Documents/scripting`, enter this:

```
export PATH=$PATH:~/Documents/scripting
```

You can add as many of these `export` statements as you need. You can also add multiple directories to your PATH in a single `export` statement by separating them with a colon (`:`), like so:

```
export PATH=$PATH:~/Documents/scripting:/Library/Scripts
```

Change Your Prompt

Your *command prompt*—the string of characters at the beginning of every command line—normally looks something like this in `zsh`:

```
jk@MacBook Pro 15 ~ % ■
```

...or like this in `bash`:

```
MacBook-Pro:~ jk$ ■
```

You can modify this by adding a line to `.zshrc` file (for `zsh`) or `.bash_profile` (for `bash`) that begins with `PS1=` and ends with whatever you want your prompt to be. For example, if you enter this:

```
PS1="I love cheese! "
```

then the next time you open a shell, your prompt looks like:

```
I love cheese! ■
```

> **Tip:** Always enclose your prompt in quotation marks, and include a space before the closing quotation mark, to make sure you can easily see where the prompt ends and commands begin.

Prompts can include variables. Some common ones are these:

- `%n` (`zsh`) or `\u` (`bash`): Your short username
- `%m` (`zsh`) or `\h` (`bash`): Your computer's name
- `%N` (`zsh`) or `\s` (`bash`): The name of the current shell
- `%~` (`zsh`) or `\w` (`bash`): The current directory
- `%D{%a %b %C}` (`zsh`) or `\d` (`bash`): The current date, in the format "Mon Feb 16"
- `%t` (`zsh`) or `\@` (`bash`): The current time, in 12-hour format with AM/PM

So, to make the following prompt:

```
jk 09:08 PM ~ * ■
```

Enter this in `zsh`:

```
PS1="%n %t %~ * "
```

Or enter this in `bash`:

```
PS1="\u \@ \w * "
```

> **Note:** In zsh, if you want the `%` character to appear by itself as part of your prompt, type it twice: `%%`.

You can even use emoji in your prompt. If you'd like it to be (or include) a pizza 🍕, baseball ⚾, sun ☀️, or some other symbol, you can paste it right into the `PS1` line of `.zshrc` or `.bash_profile` in `nano`. You can find emoji listed on many webpages or in the macOS Character

Viewer utility (available in most apps by choosing Edit > Emoji & Symbols or Edit > Special Characters).

Tip: If your command prompt (whether customized or not) shows your computer name and it's not the name you want, first try changing it in System Settings > General > Sharing or System Preferences > Sharing. If that name still doesn't appear when you open your next Terminal session, you can force a change with `scutil --set HostName "Name You Want"` and enter your password when prompted.

Tip: For another example of a profile customization, see the recipe Read man Pages in BBEdit.

Bring the Command Line into the Real World

So far in this book I've largely ignored the Mac's graphical interface, treating the command-line environment as a separate world. In fact, because the command-line interface and the graphical interface share the same set of files and many of the same programs, they can interact in numerous ways.

In this chapter, I discuss how your shell and the Finder can share information and complement each others' strengths—giving you the best of both worlds.

Get the Path of a File or Folder

Suppose you want to perform some command from the command line on a file or folder you can see in the Finder, but you don't know the exact path of that folder—or even if you do, you don't want to type the whole thing. You're in luck: there's a quick and easy way to get the path of an item from the Finder into a Terminal window.

To get the path of an item in the Finder, do the following:

1. In a Terminal window, type the command you want to use, *followed by a space*. The space is essential!

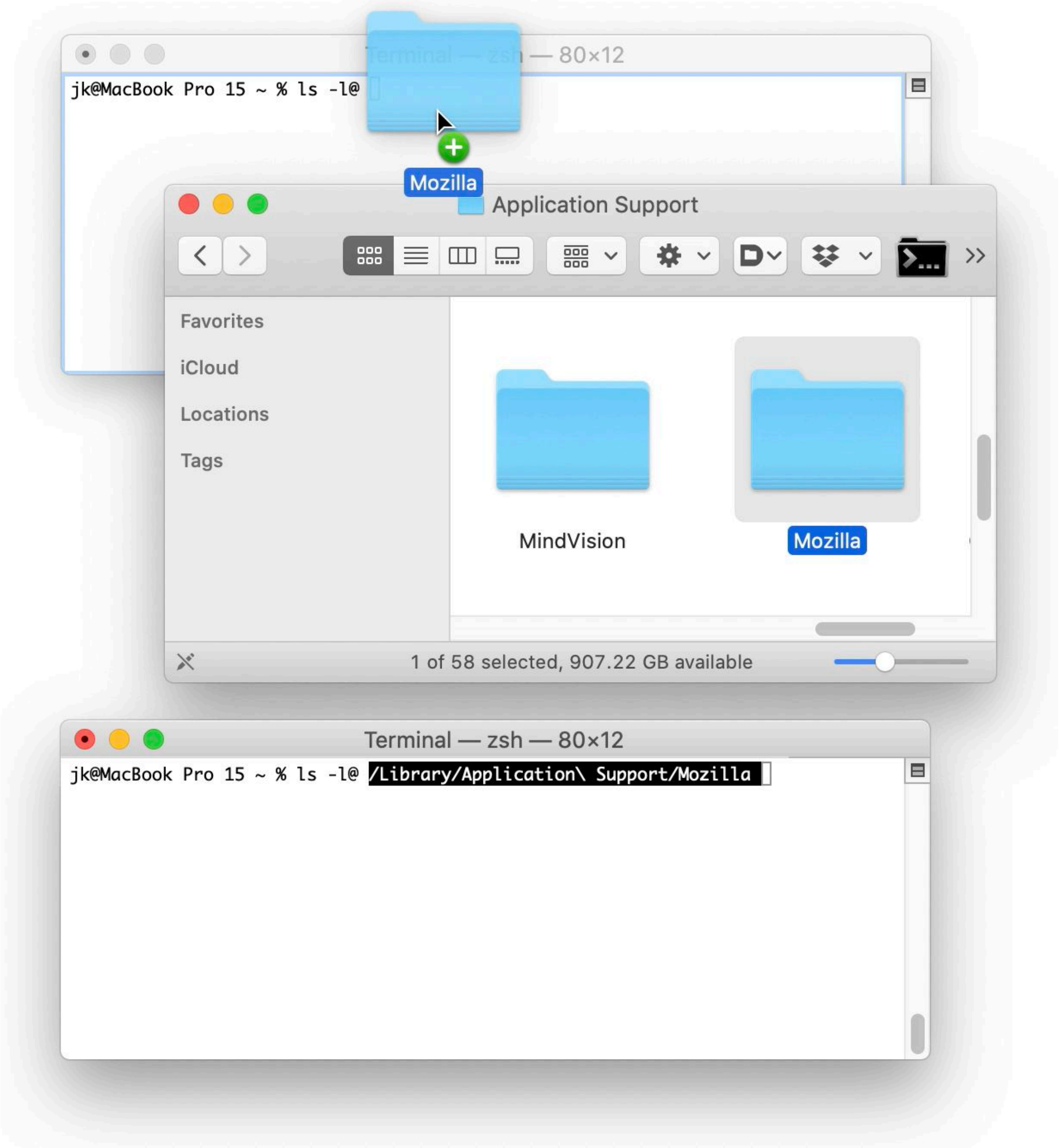
2. Drag the file or folder from the Finder into the Terminal window.

As soon as you release the mouse button, Terminal copies the path of the file or folder you dragged onto the command line. It even escapes spaces and single quotation marks with backslashes for you automatically! You can then press Return to run the command.

For example, suppose you want to use the `ls -l@` command to list the contents of a folder with their extended attributes (a type of *metadata,* or extra information about files and folders in addition to their actual contents), which you can't see in the Finder—see the sidebar What's with the + and @ Characters?. You could type this:

```
ls -l@
```

(Don't forget the space after the @!) Then drag a folder into the Terminal window, as shown in **Figure 7**.



```
jk@MacBook Pro 15 ~ % ls -l@
```

```
jk@MacBook Pro 15 ~ % ls -l@ /Library/Application\ Support/Mozilla
```

Figure 7: Drag a file or folder into the Terminal window (top); when you release the mouse button, you get that item's full path (bottom).

Open the Current Directory in the Finder

On occasion you may be using the command line deep in the macOS directory hierarchy (whether or not it's a location that's normally visible in the Finder) and want to open the current directory as a folder in the Finder.

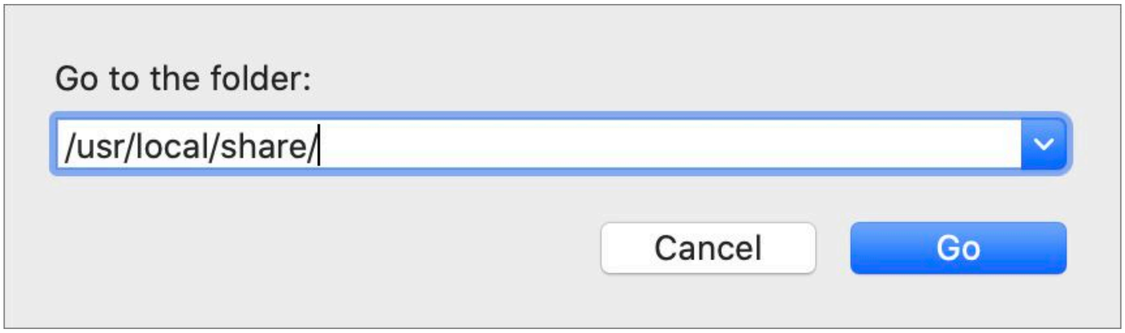
You can do so with one of the simplest commands ever:

```
open .
```

That's `open` followed by a space and a period. And that's all it takes! The single period is Unix for "the current directory"; we'll see it again later in this book.

Open a Hidden Directory without Using Terminal

If all you want to do is open a directory that's normally hidden, you need not open Terminal to do so, as long as you know its location. Just choose Go > Go to Folder in the Finder. In the dialog that appears, type the whole path of the directory (**Figure 8**) and click Go. That directory opens as a folder in the current Finder window.



Go to the folder: /usr/local/share/

Figure 8: Open almost any directory, even hidden ones, in the Finder using the Go to Folder dialog.

Tip: When you're typing a path in the Go to the Folder dialog, you can use tab completion just as in the `zsh` or `bash` shell (see Use Tab Completion); that can save you considerable typing and guessing.

Open the Current Folder in Terminal

Suppose you're looking at some folder in the Finder and you realize you want to run a command-line program on the items in it. You could open Terminal and type in the path to the folder, but that can be cumbersome. Wouldn't it be great if, instead, you could just click a button, choose a menu command, or press a keyboard shortcut and have a new shell session open in Terminal, with the current directory already set to the folder you were just looking at in the Finder?

In fact, you can do exactly that, and I'll show you two different ways to do so.

Use Services (Mavericks and Later)

Starting with 10.9 Mavericks, macOS includes two commands you can optionally add to the system-wide Services menu. One of these opens a new Terminal *window* set to the current folder, and the other opens a new Terminal *tab* set to the current folder.

To enable these, choose System Settings > Keyboard > Keyboard Shortcuts > Services or System Preferences > Keyboard > Shortcuts > Services. In the Files and Folders category, select New Terminal at Folder, New Terminal Tab at Folder, or both. (With one of these commands highlighted, you can optionally click the Add Shortcut button to add a keyboard shortcut to it as well.) Then close System Settings/System Preferences.

To use these commands, right-click (or Control-click) the folder's name. Depending on how many Services you have enabled, the New Terminal commands appear either directly on the contextual menu, or on a Services submenu. Choose the command you want to open a new Terminal window or tab at that folder's location.

Use "cd to…" (Any Version of macOS)

In any version of macOS, you can instead use the free "cd to…" utility written by Jay Tuley. Unlike the services built into Mavericks and later, this utility works even if you don't have a folder selected.

To install this utility, follow these steps:

1. Download cd to… and unzip it if necessary.
2. Drag the "cd to" app to `/Applications/Utilities` (or wherever you want to keep it).
3. While holding down Command and Option, drag the app from its new home onto the toolbar of any Finder window. (You should see a plus (+) icon appear at your pointer, signifying that the Finder is ready to add a button to your toolbar.) Move your pointer to where you want your new button to appear, and release the button.

From now on, the button (shown in **Figure 9** in Catalina) appears in the toolbar of every Finder window. You can click that button at any time to open Terminal and start a new shell session with the directory preset to your current location.

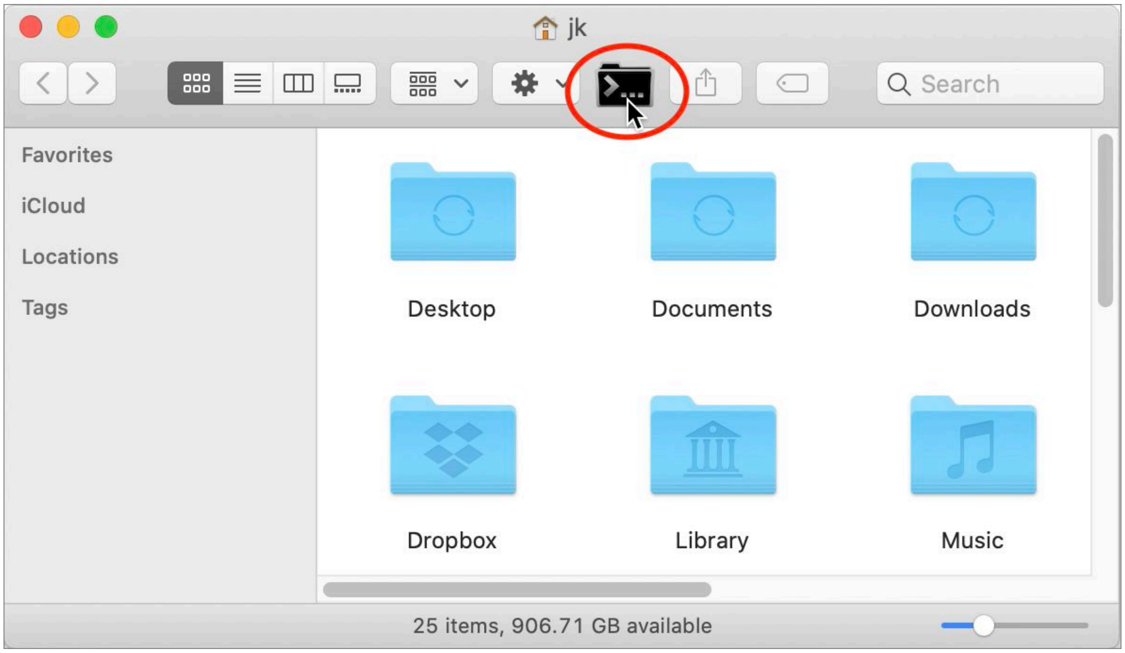


Figure 9: Click the new "cd to" button on the toolbar of any Finder window to open it in a new shell session.

Run a Shell Script from the Finder

Ordinarily, if you create a shell script (with an extension such as `.sh`) and you double-click that file in the Finder, it opens in a text editor or Xcode—whatever your default app is for files with that extension. But what if you want to *run* a shell script from the Finder? Easy: just change the extension to either `.command` or `.tool`. When you then double-click the file in the Finder, Terminal opens and your script runs. You can also drag these scripts to your Dock or open them in any of the other ways you might open an ordinary file.

Open a macOS App

If you ever need to open a regular macOS app from the command line, you can do it by entering the `open` command with the `-a` (application) flag. For example, to open Safari, just enter this:

```
open -a Safari
```

The `open -a` command is amazingly smart. You don't have to tell it where the app is located; it can be located in `/Applications`, or in `/Applications/Utilities`, or anywhere else on your disk—it doesn't matter. And you need not spell out "Safari.app" or go through any other complicated steps to get to the app.

However, occasionally, the app name you see in the Finder is not the same one that works with the `open -a` command. If the command doesn't work, select the app, choose File > Get Info, and look under Name & Extension. The name shown there should do the trick.

Open a URL

Monterey introduced another flag you can use with the open command: -u, which lets you open a URL in the default app for the URL type. For example, if your default browser is Safari, this command opens the Take Control Books homepage in Safari:

```
open -u https://www.takecontrolbooks.com/
```

Open a File in macOS

Similarly, you can open a file that you see on the command line in the default macOS app for that file type—or another app. For example, if the current directory contains a graphic named `flowers.jpg`, you can open it in its default app (probably Preview) like so:

```
open flowers.jpg
```

But if you prefer to open it in Pixelmator, just enter this:

```
open -a Pixelmator flowers.jpeg
```

Don’t forget you can use tab completion to help spell out the names of files and directories, too (but, alas, not the names of apps).

Log In to Another Computer

Every time you connect to another Mac to share files or other system resources, you are, in a way, logging in to that other Mac. However, in this chapter, I describe a particular way of logging in to a remote computer—doing so using SSH ("secure shell"), which gives you access to the other computer's command-line interface from within your own Mac's command-line interface. Logging in via SSH lets you interact with another computer in the same way you interact with your current Mac from inside a Terminal window.

You can connect to almost any Mac, Unix, or Unix-like computer (and some Windows computers) using SSH, provided the other computer has SSH enabled. (To enable incoming SSH access on a Mac, enable Remote Login in System Settings > General > Sharing or System Preferences > Sharing.)

If you log in to another Mac, everything should look quite familiar, whereas other operating systems may follow different conventions. For the purposes of this chapter, I assume that the remote computer is at least running a Unix-like system so that most of the things you've learned in this book still apply.

Start an SSH Session

The easiest way to start an SSH session from Terminal is to begin in an existing shell session. Then follow these steps:

1. Enter the following, substituting your username on the remote computer for *username*, and the remote computer's IP address or domain name for *remote-address*:

   ```
   ssh username@remote-address
   ```

2. If this is the first time you’re connecting to this particular remote computer, you will see a message something like the following:

   ```
   The authenticity of host 'macbook-pro.local
   (fe80::20c:64ce:eeb4:68ae%en0)' can't be established.
   RSA key fingerprint is
   d0:18:53:71:04:9d:c4:2e:5b:b1:f9:a0:7d:80:51:ef.
   Are you sure you want to continue connecting (yes/no)?
   ```

   After reading the sidebar “SSH Security Considerations,” just ahead, assuming you’re still comfortable connecting, type `yes` and press Return.

3. Text similar to the following appears on screen:

   ```
   Warning: Permanently added ‘macbook-pro.local.,
   fe80::20c:64ce:eeb4:68ae%en0’ (RSA) to the list of known hosts.
   ```

   And following that is a password prompt. Type your password for the remote computer and press Return.

Note: As you type your password, no text appears—not even bullet or asterisk characters. That’s normal.

Assuming the remote computer accepts your password, it presents you with a new command prompt, often (but not always) accompanied by a brief welcome message.

SSH Security Considerations

SSH is a highly secure protocol, so what’s with these fingerprints and warnings?

The simplified explanation here for using SSH relies on your trusting that the computer you’re connecting to is the one you think it is—that no one has hijacked your connection. The *fingerprint* is a unique identifier tied to each computer, and by agreeing (in step 2) that the fingerprint is correct, you’re saying you trust this fingerprint for that computer.

How would you know you can? If you’re connecting to another Mac on your home network, you can safely take it for granted. If you’re connecting to a computer at the office, a web server, or some other commercial computer, ask the system administrator who’s in charge of it to confirm its fingerprint, and make sure it matches what you see. (If the computer you’re connecting to is a Mac running Yosemite or later, you or an administrator can use the procedure in the Verify an RSA Fingerprint for SSH recipe.)

Once you accept a fingerprint, your Mac remembers it and checks to see that the fingerprint matches that remote computer every time you connect to it. If it doesn’t, it may be a sign the computer hardware has changed, or that a hacker is trying to trick you into connecting to the wrong computer.

Run Commands on Another Computer

Once you’re logged in to another computer, you run commands on it exactly the same way you do on your own Mac: just enter a command and any necessary flags and arguments.

However, you should be aware of a few potential “gotchas” when connecting to other computers:

- Your default shell on the other computer might not be `zsh` or even `bash`, so some commands may not work the way you expect. Usually—though not always—you can switch to your preferred shell, if it’s not already running, simply by entering its name.

- Your `.zshrc` file or `.bash_profile` (see Customize Your Defaults) applies only to the shell running on your Mac—*not* the shell on the remote machine! So your existing aliases, PATH variable, and other settings will not work. If you have sufficient permission, you can of course create a `.zshrc` file or `.bash_profile` on the remote computer as well.

- If the other computer is a Mac, and especially if it's running the same version of macOS that you are, you can assume that most programs will be in the same locations. But be aware that a program you want to use could be missing, located somewhere else, or configured in a way that denies you access.

- If you use a command that opens an app outside Terminal—for example, if you enter `open flowers.jpeg` to open a graphic in the default app (which on a Mac is Preview), that app opens on the *remote* computer, not the one where you physically typed the command!

End an SSH Session

To close your remote connection, simply enter `exit`.

You return to your existing shell session on your own Mac. As is the case when exiting your own shell session, it's always best to use `exit` to end a remote session gracefully, shutting down any processes that may be running and doing associated clean-up tasks.

Transfer Files with sftp or scp

Although you can run any command-line program on a remote computer while logged in with SSH, one thing you can't do in an SSH session is transfer files between your Mac and the remote computer. So, if you discover you need to move a file that's on the remote computer to your local Mac (or vice versa), you'll have to ditch SSH and use a different program. There are many that could do the trick, but I'll tell you about two of my favorites: `sftp` and `scp`.

Sftp

You've undoubtedly heard of FTP (File Transfer Protocol); you may also be aware that it's famously insecure. So even if the remote computer is running an FTP server and you have an FTP client on your local Mac, I suggest avoiding FTP as a way of transferring files unless there's no other option. But you might be lucky enough to find that the remote computer is running an SFTP (SSH File Transfer Protocol) server, which operates very much like FTP except that it's way more secure. And, as you might predict, the command you use to access an SFTP server is `sftp`.

> **Note:** Macs with Remote Login enabled in System Settings > General > Sharing or System Preferences > Sharing (that is, those you can connect to using SSH) also support file transfer via `sftp`, regardless of whether File Sharing is enabled.

To open an SFTP connection, use this command:

```
sftp username@host
```

As usual, replace *username* with your username on the remote computer and *host* with that computer's domain name or IP address. Enter your password for the remote computer when prompted. If you're connecting to a machine for the first time, you'll get the fingerprint alert described in Start an SSH Session. Then you'll see a "Connected to" message followed by this prompt:

```
sftp>
```

From here, you can use many command-line navigation techniques you're already familiar with, such as `cd` and `pwd` to browse the directories and files on the remote machine.

When you get to a directory containing a file you want to download to your local Mac, you can do it like this:

```
get filename
```

If you want to transfer an entire directory and its contents, add the `-r` (recursive) flag:

```
get -r directory-name
```

Either way, the item downloads to whichever directory you were in on the command line when you ran the `sftp` command.

Note: If the file you want isn't in the current directory but you know its exact path, you can use `get /path/to/file`. Similarly, if you want to store it somewhere else on your local Mac, you can add the destination path: `get /path/to/remote-file /path/to/local-directory`.

To *upload* a file, use the `put` command, which follows exactly the same pattern as `get`:

```
put /path/to/local-file /path/to/remote-directory
```

So, you can use just the filename if it's in your current directory, or you can specify a file from somewhere else on your Mac by giving its complete path. If you leave out the destination directory, the file is uploaded to your current directory on the remote computer.

When you're done transferring files, you can leave `sftp` by entering `exit`.

Scp

The nice thing about `sftp` is that you can use it not only to transfer files but also to *browse* the remote directory structure. But if `sftp` isn't available on the remote computer, you may have to resort to a different method: `scp` ("secure copy"). Because `scp` uses SSH, it should work pretty much anywhere SSH does, even when `sftp` does not. The downside, however, is that `scp` requires you to know the exact name and location of the file on the remote computer—you can't browse with `scp`.

If you *don't* already know the name and path of the file you want, you'll have to find that out by first logging in with SSH and browsing to find the file's location on the remote computer. Then make a note of it (or copy it to your clipboard) and switch over to `scp`—either in a separate

Terminal window or tab, or after closing your SSH connection. (As with `sftp`, if you’re connecting to a machine for the first time, you’ll get the fingerprint alert described in Start an SSH Session.)

The syntax for simple `scp` transfers is:

```
scp username@host:/path/to/remote-file /path/to/destination
```

For example, if my username on the computer `mac.alt.cc` is `joe`, the file I want to download is `/Users/joe/Desktop/test.txt`, and I want to store it on the desktop of my local Mac, I would use:

```
scp joe@mac.alt.cc:/Users/joe/Desktop/test.txt ~/Desktop
```

After you enter the command, you’ll be prompted for your password on the remote computer.

To download an entire directory, add the `-r` (recursive) flag, like so:

```
scp -r joe@mac.alt.cc:/Users/joe/Documents/Folder ~/Desktop
```

If you want to upload a file to the remote computer, you can do it almost exactly the same way as downloading, but swap source and destination, like so:

```
scp ~/Desktop/test.txt joe@mac.alt.cc:/Users/joe/Desktop/
```

And, once again, use `-r` to upload a directory and all its contents:

```
scp -r ~/Documents/Folder joe@mac.alt.cc:/Users/joe/Desktop/
```

Work with Permissions

Everything you do on your Mac, and especially on the command line, is governed by *permissions*—which user(s) can do which things with which items, under which circumstances. In this chapter, I introduce you to file permissions, along with the closely related notions of owners and groups. I also explain how to temporarily assume the power of the root user using the `sudo` command.

Understand Permission Basics

As you may recall from See What's Here, when you list files in the long format (`ls -l`), you can see the permissions, owner, and group of each file and directory. Every file in macOS has all these attributes, and you should understand how they work because they influence what you can and can't do with each item.

Before I get into how to read or change permissions, I want to describe the basic options. Put simply, permissions consist of three possible activities (reading, writing, and executing), performed by any of three types of user (the file's owner, the file's group, and everyone else). Three types of permission multiplied by three types of user equals nine items, each of which can be specified individually for any file or folder.

Read, Write, and Execute

Someone with permission to *read* a file can open it and see what's inside it. Someone with *write* permission can modify an item or delete it. *Execute* permission, for a file, means it can be run (that is, it can behave as a program or script); for a directory, execute permission means someone can list its contents.

On the command line, read permission is abbreviated with an `r`, write permission is abbreviated with a `w`, and execute permission is abbreviated with an `x`.

User, Group, and Everyone Else

Every file and folder specifies read, write, and execute permissions for the following types of user:

- **User:** In terms of file permissions, the term *user* means the owner of a file or directory. (The user may be a person, like you, or it may be a system process, such as `_screensaver`, which is exactly what it looks like.)

- **Group:** Each file and directory also has an associated group—one or more users for whom a set of permissions can be specified. That group could have just one member (you, for example), or many. macOS includes several built-in groups, such as `admin` (all users with administrator access), `staff` (all standard users without administrative access), and `wheel` (which normally contains only the root user—see Perform Actions as the Root User). You can also create your own groups.

- **Others:** Every user who is neither the owner nor in the file's group is lumped into the "others" category.

Reading Permissions, Owner, and Group

To illustrate how this all works, suppose you find the following two items in a certain directory by entering `ls -l` ("list in long format"):

```
drwxr--r--  15 jk    admin  510 Aug 27 15:02 fruits

-rw-r--r--  2  root  wheel 1024 Sep 02 11:34 lemon
```

For the purposes of this section, we care about just three of the items on each line (apart from the item's name, at the end). The initial group of characters (like `drwxr--r--`) constitutes the permissions, and the two names in the middle (like `jk admin`) are the user and group, respectively. For now, you can ignore all the other information.

Directory or Not?

The first character of the permissions string tells you whether the item in question is a directory or a regular file. So in the first example (`drwxr--r--`), the item `fruits` is a directory because its permissions string starts with a `d`. The second item, `lemon`, has a hyphen (`-`) in the first slot, which means it's not a directory (in other words, it's a file).

Three Permissions, Three Sets

The remaining nine positions in the mode specify the three possible permissions for user (the first three characters), the group (the middle three), and others (the final three).

In each set of three characters, the order is always the same: `r` ("read"), `w` ("write"), and `x` ("execute"). So picture a template with ten slots, of which the first is the `d` character for directories:

`directory`	`user`	`group`	`others`	← Access for whom
`d`	`rwx`	`rwx`	`rwx`	← A directory with all attributes on
`-`	`---`	`---`	`---`	← A file with all attributes off

For each kind of user, each permission can be either on or off. If it's on, the corresponding letter (`r`, `w`, or `x`) appears; if it's off, you see a hyphen (`-`). So, for example, if the owner's permissions are `rwx`, it means the owner can read, write, and execute the item; if they're `r--`, the owner can only read the item.

If everybody—user, group, and others—had read, write, and execute permissions for a file, its permissions would look like this:

```
-rwxrwxrwx
```

Here are a few other combinations to make the system clear:

- Owner can read, write, and execute; group and others have no permission:
  ```
  -rwx------
  ```
- Owner can read and write; group and others can read:
  ```
  -rw-r--r--
  ```
- Everyone can read and execute, but only the owner can write:
  ```
  -rwxr-xr-x
  ```

- Owner can read and write; group can read only; others have no permission:
  `-rw-r-----`

Owner and Group

After the file's permissions and a number (the number of links to the item—a concept I touched on briefly in See What's Here) are two names. The first of these is the file's owner (user) and the second is its group.

For example in this item:

```
drwxr--r--  15 jk   admin   510 Aug 27 15:02 fruits
```

the owner is `jk` and the group is `admin`. (In some cases, owner, group, or both may be shown as numbers, such as `501`, rather than names.)

What's with the + and @ Characters?

Sometimes a file has an extra character at the end of the permissions string—either a `+` or an `@`. For example:

```
drwx------+ 90 jk  staff  3060 Aug  1 09:29 Library
drwx------@  8 jk  staff   272 Jul 11 11:24 Movies
```

The `+` means the item includes an ACL (access control list), which is a more elaborate and finer-grained way of specifying permissions than simply read, write, and execute for user, group, and others. To see the ACL settings for a file or directory, use `ls -le`.

The `@` means the item includes extended attributes—extra metadata beyond the file's contents often used for specific macOS features (such as Gatekeeper). To see which types of extended attributes a file or directory contains, use `ls -l@`; to view the contents of those extended attributes, use `xattr -l file`.

Understanding, using, and modifying ACLs and extended attributes is, alas, beyond the scope of this book. To learn more about ACLs, see Apple's article Access Control Lists or use `man chmod` and scroll down to the topic "ACL Manipulation Options." For more on extended attributes, see the article Show me your metadata or try `man xattr`.

Permissions and You

When you create a file (whether by saving, copying, moving, downloading, or whatever), you become that file's owner (user).

In addition, by default, all users on a Mac have read and write permission (and, for directories, execute permission) for everything in their home folders, and can read and execute shared items (such as things in the `/Applications` folder). However, users can't read or write files, or view the contents of directories, owned by other users.

Your default group (and thus, the default group of files in your home folder and new items you create anywhere) depends on a few factors, the most significant of which is what sort of user account you're using. Account types are specified in System Settings/System Preferences > Users & Groups (called Accounts in earlier versions of macOS). If you're an administrator, your default group is normally `admin`; otherwise, it's normally `staff`.

Change an Item's Permissions

If you want to change an item's permissions, you use the `chmod` command (for "change mode," *mode* being a Unix way of describing an item's permissions). You can use `chmod` in a number of ways. The easiest one to understand is what you may sometimes hear described as `chmod`'s *symbolic* mode. There's also a more-powerful *absolute* mode, which we'll get to in a moment.

Use the chmod Symbolic Mode

To change permissions with `chmod`, you indicate one or more of user, group, and others (using the abbreviations `u`, `g`, and `o` respectively), then `+` or `-` (to add or remove permissions), and one or more of `r`, `w`, and `x` (for read, write, and execute), followed by the file or directory. For example, to grant group write access to the file `file1`, you might enter this:

```
chmod g+w file1
```

To remove others' execute permission, enter this:

```
chmod o-x file1
```

You can affect multiple users at once—for example, add read access for user, group, and other in one stroke with this:

```
chmod ugo+r file1
```

You can also affect multiple permissions at once; for example, subtract read, write, and execute permission for the group and others with the following:

```
chmod go-rwx file1
```

Note: Ordinarily, you can change an item's permissions only if you are the owner or are in the item's group, and if you already have (in either capacity) write permission. In all other cases, you must use `sudo` (described ahead) before the `chmod` command.

Use the chmod Absolute Mode

In order to make more complex changes in one go (say, adding write permission for the user while removing execute permission for others), you must use `chmod`'s *absolute* mode. This is somewhat advanced, but as you work on the command line you're bound to come across it, so I want you at least to be familiar with how it works.

In absolute mode, permissions are indicated by a series of three digits, such as `133` or `777`. The first of these digits stands for the user, the second for group, and the third for others (just as in symbolic mode). But discerning the meanings of the numbers requires a little arithmetic.

The basic values are these:

- `4`: read
- `2`: write
- `1`: execute

To combine permissions, you add these numbers. So, `6` means read and write; `5` means read and execute (but not write); `3` means write and execute; and `7` means read, write, and execute.

Thus, if you read an article telling you to change a file's permission with this command:

```
chmod 755 file
```

it means you want the user to be able to read, write, and execute, while the group and others can read and execute only. In other words, the file's permissions would become:

```
-rwxr-xr-x
```

Change an Item's Owner or Group

To change an item's owner, group, or both, use the `chown` ("change owner") command. It takes one or two arguments—the new owner and/or the new group, separated by a colon (`:`)—followed by the item you want to change. For example, to change the owner of the file `file1` to `bob` (without changing the group), enter:

```
chown bob file1
```

To change the owner of file1 to `bob` *and* the group to `accounting`, enter:

```
chown bob:accounting file1
```

To change *only* the group, but not the owner, simply leave out the owner but include the colon before the group:

```
chown :accounting file1
```

However… What I just said is hypothetical, because as an ordinary user you can't change an item's owner—that would mean changing it either to or from an account to which you don't have access! Similarly, you can change an item's group only if you're a member of both the old group and the new group. So for all practical purposes, the `chown` command must always be performed using `sudo`, described next.

Perform Actions as the Root User

As a security measure, macOS (like all Unix and Unix-like operating systems) prevents users from viewing or altering files that don't belong to them, including those that make up the operating system itself. However, in certain situations you may have a legitimate need to alter a file or folder of which you're not the owner—or run a command for which your user and group don't have execute permission.

Every Mac has a special, hidden account called `root`, which is a user with almost unlimited power (subject to certain constraints starting with 10.11 El Capitan, as I describe just ahead). The root account is disabled by default, and that's for the best. However, any administrator can *temporarily* assume the capabilities and authority of the root user, even without the root account as such having been activated.

The way you do this is to use the `sudo` ("superuser do") command.

Note: Because the "do" in `sudo` is the actual verb *do*, the preferred pronunciation of the term rhymes with "voodoo." But lots of people pronounce it to rhyme with "judo," which is also logical—and it's acceptable to everyone except the nitpickiest geeks.

Prior to El Capitan, using `sudo` or logging in as the root user enabled someone to change virtually anything on the computer. But in El Capitan, Apple introduced a security feature called System Integrity Protection, which prevents certain critical files and directories—including `/System`, `/bin`, `/sbin`, `/usr` (except for the `/usr/local` and `/usr/share` subdirectories), and certain critical apps—from being altered in any way, even with the `sudo` command; only the Apple installer itself can now change those items. Starting in Catalina (see The Startup Volume Became Read-Only), Apple made the entire system volume read-only, which takes this idea even further. This restriction is unlikely to affect anything a normal person would want to do on the command line, but if you try a `sudo` command and it fails, that could be why.

Another requirement to use `sudo` (in Mojave and later)—for *certain* commands only—is granting Terminal Full Disk Access in System Settings > Privacy & Security > Full Disk Access or System Preferences > Security & Privacy > Privacy Full Disk Access; see the sidebar Special Permissions for Terminal and the Command Line. And, if you're using Catalina or later, recall that the core components of macOS are now stored on a read-only volume (see The Startup Volume Became Read-Only), which means they're completely immune to change, with or without the use of `sudo`.

For Administrators Only

Before I go any further, I must make this crystal clear: only users with administrator privileges can use `sudo`. If your Mac has just one user account, it's automatically an administrator account. However, as you create additional accounts, they only gain administrator privileges if you check the Allow User to Administer This Computer box in the System Settings/System Preferences > Users & Groups.

Most Mac experts recommend using a *non*-administrator account for ordinary, day-to-day computing, logging in as an administrator only when necessary.

That's good advice, but if you follow it, you'll have to do one of two things before you can make use of the `sudo` command:

- Log in as an administrator first, and then run Terminal, *or*
- In your shell session in Terminal, switch to an administrator's account using the `su` ("switch user") command, like so:

  ```
  su username
  ```

  (Replace *username* with the short username of an administrator, and enter that account's password when prompted.)

Note: As you type the administrator account's password, no text appears—not even bullet or asterisk characters. That's normal.

It’s a good idea to keep excursions to other accounts brief. When you’ve finished executing commands as another user, you can use the exit command to return to your normal account.

Using sudo

Once you’re logged in as an administrator, to perform any command as the root user, preface it with sudo:

```
sudo command
```

The sudo command prompts you to enter the administrator account password; do so now.

Note: As you type your password, no text appears—not even bullet or asterisk characters. That’s normal.

The shell then performs whatever command you just entered as though you’d entered it as the root user, which ordinarily means it’s guaranteed to work as long as you entered it correctly.

Note: The first time you use sudo for certain operations under Mojave or later, you may be prompted to grant Terminal special permission in System Settings > Privacy & Security or System Preferences > Security & Privacy; see Special Permissions for Terminal and the Command Line for details.

Tip: If your Mac has a Touch ID sensor, you can use your fingerprint to authenticate sudo instead of typing your password! This article by Dan Moren at Six Colors has instructions for setting that up, and a follow-up article explains how, starting in Sonoma, that change can persist through system updates.

If you perform a command and get a “permission denied” error, try it again with sudo in front of it, and in all probability it will work the second time (bearing in mind the sudo limitations I mentioned earlier).

For example, if you try to change a file’s owner like so:

```
chown bob file1
```

and you get this message:

```
chown: file1: Operation not permitted
```

try this instead: `sudo chown bob file1`

Tip: Now that you understand how `sudo` works, you may enjoy this highly geeky comic from xkcd: Sandwich.

As a reminder (refer back to the sidebar !! Plus), if you try a command and it doesn't work, rather than retype the whole thing with `sudo` in front of it, you can use this shortcut:

```
sudo !!
```

That means "Do the previous command again, but as root." This is especially helpful with long, complex commands.

Notes and Precautions

Before you start using `sudo`, you should be aware of a few things:

- **5-minute rule:** Once you use `sudo` and enter your password, you can enter additional `sudo` commands, *without* being prompted for a password, for 5 minutes. The timer resets every time you use `sudo`.
- **Great power = great responsibility:** You can do almost anything with `sudo`, and that includes damaging macOS beyond repair (although this is less likely in 10.11 El Capitan and later, due to System Integrity Protection—and less likely still starting in Catalina, due to the read-only system volume). So use `sudo` only when necessary, and only when you know what you're doing.
- **Stay for a while:** If you must enter numerous commands with root privileges, you can avoid having to enter `sudo` every time by switching to the root user's shell account (even if the root account isn't enabled on your Mac). To do so, enter `sudo -s` and supply your password if requested. Your prompt changes from a `$` to a `#` to signify that all commands you enter are performed as the root user.

Be extra careful! If `sudo` alone is dangerous, `sudo -s` is asking for trouble. It’s a convenience feature I personally use on rare occasions, and it can be handy in a few situations in which `sudo` alone won’t do the trick. But use this with the utmost caution, and be sure to use the `exit` command to log out of the root user’s shell as soon as possible.

Learn Advanced Techniques

Now that you know the basics of working with the command line, I want to show you a few techniques that build on your knowledge and enable you to perform more advanced tasks.

First I tell you how to Pipe and Redirect Data—two powerful (and related) techniques you can apply to many different commands in order to combine them in useful ways and do more with your data. Next, you'll Get a Grip on grep, a tool that helps you locate files containing specified patterns of characters. Finally, I explain the basics of how you can Add Logic to Shell Scripts, making them much more useful than simple sequences of commands.

As you can imagine, these are but a few of many advanced techniques for using the command line, but I've found them to be consistently helpful, and I hope you will too.

Pipe and Redirect Data

Most of the time when you enter commands on the command line, the output—a list of files, the date, the contents of a log, or whatever—is shown directly on the screen. But that isn't always what you want.

For example, maybe the output of some command is a list of hundreds or thousands of files, but that's more information than you need; you want to filter the list to show only files that meet certain criteria. Or, maybe having that list in a Terminal window isn't useful to you, but if it were in a BBEdit document, it would be. In cases like these, you can use either of two commands to take a command's output and do something other than display it on the screen. You can also use a slightly different command to redirect a command to use a different input.

Pipe (|)

The pipe operator, which is the `|` symbol that you get when you type Shift-\, sends the output of a command to another *program*. To use it, you type the first command, then an optional space, the `|` character, another space, and the name of the second program. Like so:

```
program | other-program
```

We saw the pipe earlier, in Ps, and there are also a few instances of this in Command-Line Recipes, but let me give you some further examples to illustrate how this works and what you might do with it.

If I used the `ls /Library/Preferences` command to show me everything in my `/Library/Preferences` folder, that would be a pretty long list. But suppose I remembered that most of the items in that folder started with `com.apple` and I wanted to see just the last, say, 10 items because that would filter out most of the Apple stuff. And then I remember that the Tail command does exactly that. Ordinarily, `tail` expects you to give it a *file* as an argument. But instead, I could give it a file *listing* as an argument, using the pipe operator, like so:

```
ls /Library/Preferences | tail
```

And that does what I expect—it shows just the last 10 items from that directory. If I wanted to show the last 15, I could instead enter:

```
ls /Library/Preferences | tail -n 15
```

Most flags and arguments work as usual with piped commands. The exception: commands expecting a file as an argument normally put the file *after* the command, but when you use a pipe, the order is reversed.

How about another example? If I used the `locate` command to find all the files containing *Apple* in the name—again, an awkwardly large number—they'd all scroll by at a dizzying speed. If instead I wanted to be able to page through them one screenful at a time—hey, just like you can do with `less` (see View a Text File)—I can just pipe the output of `locate` into `less`, like so:

```
locate Apple | less
```

Or perhaps I'd like to get the path of the current directory and put it on my macOS clipboard. With a pipe and the `pbcopy` ("pasteboard copy") command, it's easy:

```
pwd | pbcopy
```

The same idea works for other commands. Need to copy a list of every GIF image in a directory? Entering `ls *.gif | pbcopy` does the trick.

These examples are all fairly simple, but the concept can be extended in all kinds of ways. If a command can accept a file as an argument, it can probably be used on the right side of a pipe.

And, in case you were wondering, yes, you can *chain* pipes! That is, send the output of one program to a second, and the output of the second to a third (and so on). So, if I want my clipboard to contain a list of the last ten files in my `/Library/Preferences` directory (without displaying them on the screen), I could combine a couple of earlier example like so:

```
ls /Library/Preferences | tail | pbcopy
```

This is a technique that rewards experimentation, so see what other interesting combinations you can come up with.

Tip: The `pbcopy` command has a counterpart that pastes onto the command line whatever's on your Mac's clipboard: `pbpaste`.

Redirect Output (>)

Whereas the pipe sends the output of a program to another program, the redirect output (`>`) operator sends the output of a program to a *file* (without displaying it on screen). For example, maybe I want to put the list of all the files in `/Library/Preferences` into a text file to study later. I could do it like this:

```
ls /Library/Preferences > ~/Desktop/prefs.txt
```

That creates a file on my desktop called `prefs.txt` which contains the output of the `ls` command—it lists everything in that directory.

You can use redirect with nearly any (non-interactive) program that displays its output on screen. But be careful with commands that produce continuous output; the file will keep growing indefinitely. For example, you wouldn't want to use `top > file.txt` because the top command produces a dynamic output. Instead, you might try `ps -ax > file.txt` for a static snapshot of all running processes.

> **Tip:** If you want to *add* to an existing file using redirect, rather than *replace* its contents, use `>>`. For example: `ps -ax >> file.txt`

At this point, perhaps your gears are turning and you want to know whether you can combine piping and redirecting in a single line. Why, yes, you can! If I wanted to put the last 10 items from a directory into a text file in my home folder, I could do it like this:

```
ls /Library/Preferences | tail > ~/files.txt
```

Redirect Input (<)

Sometimes it's helpful to do the reverse of redirecting output—you may have to redirect *input*. To illustrate what this means and one way to do it, let's go back to the `pbcopy` command we used in the Pipe (|) discussion. You can pipe output of another process to `pbcopy`, but what if you want `pbcopy`'s input to be a *file* rather than the output of another command? How might you accomplish that?

Well, for a simple text file, you could do something like this:

```
cat file.text | pbcopy
```

However, there's an easier way to accomplish that task—by using the redirect input (`<`) operator:

```
pbcopy < file.txt
```

That reads: redirect the *input* of `pbcopy` to `file.txt`. It's more efficient because it uses just one program rather than two.

Get a Grip on grep

Depending on who you ask, the command grep stands for either "*globally* search a *regular expression* and *print*" or "global regular expression parser." In any case, grep is a pattern-matching tool that can make use of a sequence of characters known as a *regular expression* (sometimes abbreviated to regex or regexp) in order to locate files by their content. If you know what you're looking for inside a file but not the file's name or location, this is the command you want.

We'll get back to regular expressions in a moment. First, let's look at a very basic use of grep that uses a plain text search string.

Earlier, in Find a File, I showed how to use the find command to find a file by name. It's also possible to find a file with find based on the file's content, but an easier way is to use the grep command. Enter the following, replacing *your text* with what you want to find:

```
grep -R "your text" .
```

For example, to find all files within the current directory and its subdirectories whose contents (not necessarily filenames) include the word *Apple,* I'd use:

```
grep -R "Apple" .
```

The -R flag means "recursive"—that is, look in all the subdirectories. Also notice the period (.) at the end. That signifies "this directory." So the combination of -R and the period mean "search recursively from this directory down." To search just the directory you're in, you can leave out -R, but then you must also replace the period with an asterisk (*), to mean "any file"—without that, grep indicates an error because you've told it to search a directory, but it searches only *files*.

If I wanted to search recursively from the parent directory of the one I'm in, I'd do this:

```
grep -R "Apple" ..
```

Those are the same two dots (`..`) we used with `cd` (see Move Up or Down). And if I wanted to search a specific directory, I'd fill in its path:

```
grep -R "Apple" /Library/Preferences
```

I suggest resisting the temptation to put `/` (a whole disk) as the search target, because the search would be enormously time-consuming.

> **Note:** By default, `grep` finds partial-word matches; the string `"bar"` matches both *baroque* and *lumbar*.

That's the simple way to use `grep`, and it's pretty useful. But what if you're not looking for a specific string, like *Apple,* but rather a *pattern,* such as a phone number, a URL, or any line that starts with the word *butter*? That's where regular expressions come in.

A regular expression is basically a pattern of regular characters and *metacharacters* (such as wildcards, parentheses, and other special symbols that tell `grep` to look for particular characters or patterns of characters). With practice, you should be able to create a regular expression that represents almost any text you can describe in words. Here are some simple metacharacters to get you started; keep reading for how to incorporate these expressions into the `grep` command:

- Any character: `.` (period)
- One or more times: `\+`
- Anything in a particular set of characters: `[ ]` (for example, `[abcde]` for any of the letters a, b, c, d, or e; or `[1-5]` for any digit 1 through 5)
- Start of a line: `^`

So, putting various combinations of these together, we can look for:

- **Any seven-digit phone number:**

  ```
  [0-9][0-9][0-9]\-[0-9][0-9][0-9][0-9]
  ```

  Bracketed sets of characters can include ranges, like `[0-9]`, `[A-Z]`, or `[a-z]`. Because some characters, like `-`, have special meanings in regex, you put a backslash (`\`) before them to indicate that you're looking for the literal character here.

- **Any number of digits followed by a hyphen and any number of additional digits:**
  `[0-9]\+\-[0-9]\+`

- **Any instance of the word *butter* at the beginning of a line:**
  `^butter`

- **Any number with three or more digits at the beginning of a line:**
  `^[0-9][0-9][0-9]\+`

- **Any line that starts with a digit or an uppercase letter:**
  `^[0-9A-Z].\+`

  You can combine ranges of characters in a set. And `.+` means "any character, one or more times."

That's just the beginning. There are metacharacters to represent all kinds of things. A few more examples:

- Anything *not* in this set of characters: `[^ ]`
- A space: `\s`
- A tab: `\t`
- End of a line: `$`
- A return character: `\n`

You can also group elements in parentheses `( )`, use the pipe `|` character to indicate "or," and much more. (There are many different versions of regular expressions, which vary in which metacharacters they support.)

Now let's go back to searching for files by content, because that's what kicked off this topic. Let's say I'm looking for any file that contains the word *Apple* as the *next-to-last* word of a line. I start with the regular expression:

```
Apple\s[A-Za-z]\+$
```

That reads "the word Apple, followed by a space, followed by any string of one or more uppercase or lowercase letters, at the end of a line." Now I feed that to grep, like this:

```
grep -RIE "Apple\s[A-Za-z]\+$" .
```

Notice the two new flags: -E means "treat this as a regular expression, not plain text," and -I means "ignore binary files" (since I know I'm searching only for matching text files, this makes the command run much faster by skipping things like image, audio, and video files, but also ignores things like Microsoft Word documents and PDFs).

Needless to say, you can also combine grep with other commands using piping and redirecting (as discussed in the previous topic). For example, to list all the files in a directory but show only those containing the word *Apple,* you might try:

```
ls /Library/Preferences | grep -ERI "Apple"
```

All this is still the tip of the iceberg. Regular expressions are useful not just in grep but also in Perl scripts and in macOS apps such as Nisus Writer Pro and BBEdit. And grep can do far more than what I've described here.

Note: To learn more about grep, read Kirk McElhearn's Macworld article Find anything with grep, and to learn more about regular expressions more generally, read Jason Snell's article Transform HTML with Regular Expressions. You might also enjoy reading Linux/BSD command line wizardry: Learn to think in sed, awk, and grep by Jim Salter, which covers grep as well as sed (used to make text replacements) and awk (used to find and replace text in columns).

Add Logic to Shell Scripts

When I showed you how to Create Your Own Shell Script, the examples I gave were simple sequences of commands: do this, then this, then this; and you're done. But sometimes you may need scripts to be more flexible. They might need to accept input, make decisions, perform calculations, and employ other sorts of logic.

If you've done any type of programming or scripting, you've certainly encountered concepts like variables, conditionals, and loops. You can use all these things in `zsh` and `bash`, too, although you'll need to learn the idiosyncratic way these shells deal with them. Alternatively, if these concepts are brand new to you, shell scripting is one of the easiest ways to learn by experimentation.

My intention here is not to teach you programming or provide extensive tutorials, but only to provide a few simple examples to get you started, along with some pointers to places where you can learn more.

Variables and Input

In `zsh` and `bash`, variables are about as simple as they get in any programming language. You can pick almost any word you like to serve as a variable, and you give it a value by typing `=` and a number or *string* (any sequence of characters). For example, if I want a variable called `city`, I can create it and give it the value `12345` like so:

```
city=12345
```

Or, if I want it to have the value *New York,* I do it this way:

```
city="New York"
```

I put New York in quotation marks because it has a space in it. If the string didn't have a space, I could have left out the quotation marks, but using them with strings is a good habit to get into, because multiword strings are pretty common. Other than that detail, you don't need to do anything special to tell `zsh` or `bash` whether a variable is an integer or a string—a distinction that's important in most other programming languages.

Note: You'll notice that there are *no spaces* around the `=` sign. This is crucial: if you used spaces (as in `city = 12345`), the shell would mistakenly think that the variable name is the name of a command, and the script wouldn't work.

Later on, if I want to do something with my variable, such as display it on the screen, use it in a computation, or compare it to another value, I put a dollar sign (`$`) in front of it. For example, this (rather pointless) script assigns a value to a variable and then displays it:

```
#!/bin/zsh
city="New York"
echo $city
```

I'd like to show you three additional tricks with variables, two of which involve getting input from the user.

Turn a Command-Line Argument into a Variable

We've seen a lot of commands that take arguments. For example, the command `nano file.txt` opens the file `file.txt` in the `nano` editor, and `ls /Library` lists the contents of the `/Library` directory. You can do the same thing with your own scripts: add one or more arguments after the script's name to provide more information to the script about what you want it to do. Best of all, it requires almost no effort.

When you enter a script name followed by a space and one or more terms, each term is automatically assigned to variables called `$1`, `$2`, `$3`, and so on in the order the terms were typed. For example, suppose we created this script and named it `test.sh`:

```
#!/bin/zsh
echo "The first three arguments you entered were $1, $2, and $3."
```

Now run the script like so:

```
./test.sh Alice Bob Carol
```

The output will be:

```
The first three arguments you entered were Alice, Bob, and Carol.
```

If you entered more than three arguments, the rest will be ignored (although you could add `$4` to the script easily enough), and if you entered fewer, the response would have some blanks, as in:

```
The first three arguments you entered were Alice, , and .
```

(And yes, you could add logic to the script to eliminate those blanks, but I'm trying to keep things simple for now.)

Note that anything you can type on the command line can be a variable, including pathnames and filenames.

Tip: If you need your script to know its own name for any reason, that's easy—it's stored in a variable automatically: `$0`.

Get Interactive User Input

You can also have the script ask you a question *while it's running* and turn your response into a variable. You do that with the command `read` followed by the name of the variable you want the response to be stored in:

```
#!/bin/zsh
echo "What do you have to say for yourself?"
read reply
echo "Oh yeah? Well, $reply to you too!"
```

A script can carry on an extensive conversation with the user, if need be, and each response can influence what happens later in the script.

Put the Output of a Command into a Variable

The last variable trick I want to mention is useful when your script needs to run a command and then do something with that command's output. For example, if you use the `date` command to find the date, you may want to put the date into a variable so that you can later use it as part of a filename. Or if your script uses the `pwd` command to find the path of the current directory, you might want to use that information later on when saving a file.

To do this, surround the command in question (including any flags or arguments) in parentheses, with a dollar sign `$` before them, as in:

```
today=$(date)
```

or

```
directory=$(pwd)
```

Then, later on you could refer to `$today` or `$directory`, respectively, to retrieve the contents of those variables.

Flow Control

Scripts frequently make decisions based on user input or information they encounter as they run. For example, let's say you have a script that renames the files in a directory, but you want to rename them one way if they're text files, a different way if they're PDFs, and a third way if they're PNG graphics. Or suppose you want to ask the user for a number and perform one action if the number is less than or equal to 5, but a different action if the number is higher.

In cases like these, you need conditional statements like *if*, *then*, and *else*. These are sometimes called *flow control* statements, because they determine the path the script takes.

The `zsh` and `bash` shells have a weird way of structuring if/then statements. Here's the basic structure:

```
if [ condition to test ]
then
   action to take
fi
```

I want to point out a few key items here:

- The condition in the first line (a mathematical or logical test that yields a true/false result) must be surrounded by spaces inside the brackets. (Remember that you can't have spaces around `=` in variable assignments; here, they're mandatory.)

- After the `if` line containing the condition, you need the word `then`—either on a line by itself (as above), or on the same line after a semicolon (as I show in the next example).

- By convention, most people indent the command(s) that follow `then` by a few spaces or a tab, but that's just to make your script easier to read. You can leave them out if you prefer.

- Every if statement must end with `fi` (that's `if` backward), which is equivalent to `end` or `endif` in other languages.

Here's a complete script that shows how `if` works:

```
#!/bin/zsh
echo "Pick a number."
read reply
if [ $reply -le 5 ]; then
   echo "$reply is less than or equal to 5"
fi
```

(We'll get to that funny `-le` thing in a minute.)

But wait… what if the number is greater than 5? Then you need to expand the `if` statement to include `else` (what to do if the condition presented is false).

You do it like so:

```
#!/bin/zsh
echo "Pick a number."
read reply
if [ $reply -le 5 ]; then
   echo "$reply is less than or equal to 5"
else
   echo "$reply is greater than 5"
fi
```

You can check for two or more conditions, too. For example, do one thing if the number is less than 5, a second thing if the number is exactly 5, and a third thing if the number is greater than 5.

To do this, you'll add `elif` ("else if"), along with another `then`, like this:

```
#!/bin/zsh
echo "Pick a number."
read reply
if [ $reply -lt 5 ]; then
   echo "$reply is less than 5"
elif [ $reply -eq 5 ]; then
   echo "$reply is exactly 5"
else
   echo "$reply is greater than 5"
fi
```

Well, what about that funny `-le` in the first example, or the `-lt` in the last one? Those mean *less than or equal* and *less than,* respectively. Wacky, I know, but `zsh` and `bash` don't use symbols like `≤` or `<=`, relying instead on abbreviations for the most part. Here's a longer list of operators you might need to know:

- Is greater than: `-gt`
- Is less than: `-lt`
- Is equal to (for integers): `-eq`
- Is equal to (for strings): `=` (see Note below)
- Is not equal to: `!=`
- Is greater than or equal to: `-ge`
- Is less than or equal to: `-le`
- Contains a string (not integer or empty): `-n`
- Is empty: `-z`
- Logical AND: `&&`
- Logical OR: `||`
- Logical NOT: `!`

Be especially careful with those “is equal to” operators, because if you use the wrong one for the type of thing you’re comparing, you’ll get the wrong results (or an error message). For example, if `$this` is a number, you might have `if [ $this -eq 5 ]`, but if `$this` is a string, you would have to use `if [ $this = "Joe" ]`.

Note: In `bash` scripts, you can use `==` in place of `=` to compare strings, but in `zsh`, you must use a single `=`.

Loops

In order to perform an operation on every file in a directory, every line in a file, or every *whatever* of a *something*, you need a loop. As in most programming languages, `zsh` and `bash` offer several loop varieties. Here’s how they look.

While Loops

If you need to repeat an action as long as some condition is true (or *while* it’s true) but then stop when it becomes false, you want a *while* loop. The structure is as follows:

```
while [ condition to test ]
do
    stuff to do
done
```

For example, this while loop displays the numbers from 1 to 10:

```
#!/bin/zsh
count=1
while [ $count -le 10 ]
do
   echo "$count"
   ((count++))
done
```

Note: Like `if`/`then` statements, the `do` can go on its own line, or on the same line as `while`, separated with a semicolon.

We start by saying that the `$count` variable is the integer 1, and each time through the loop we display its current value and then add 1. That's what the `((count++))` line does—the double parentheses mean "this is a mathematical operation" and the `++` means "add 1."

For Loops

A *for* loop starts with a list, series, or range of items (numbers, files, etc.) and performs one or more actions once for each of those items. Its basic structure is:

```
for variable in list
do
   stuff to do
done
```

As an example, here's a simple script that displays five consecutive messages, each with the number of the current iteration:

```
#!/bin/zsh
for i in 1 2 3 4 5
do
   echo "This is iteration number $i"
done
```

You can also represent a range using brackets, as in `{1..5}` (notice that there are just two periods in between the numbers, not three). And the items don't have to be numbers—they can be anything. For example:

```
#!/bin/bash
for i in Red Orange Yellow Green Blue
do
   echo "$i is a lovely color."
done
```

If an item in a range includes a space, you must escape the space by putting a backslash before it:

```
for i in New\ York Seattle San\ Francisco
```

Math

When it comes to math, `bash` is at about first-grade level, while `zsh` is a bit better out of the box and can be extended even further with optional modules. However, both shells can do mathematical basics like add, subtract, multiply, divide, and compare integers (whole numbers). You can use external calculators (such as `bc`) in a `bash` script to perform more advanced calculations.

As we saw in While Loops, you can tell `zsh` or `bash` that you want it to calculate something by surrounding it with double parentheses:

```
((7*5+3))
```

But if you want to do anything with that result, such as assign it to a variable, you must add a `$` to the beginning, which in `zsh` and `bash` is known as *arithmetic expansion:*

```
number=$((7*5+3))
```

A different way to achieve the same result is to use the `let` command, which also requires quotation marks around the entire expression, including the variable, like this:

```
let "number=7*5+3"
```

Tip: As a reminder, you can retrieve variables later by prefixing them with a `$`—for example: `echo $number`.

Note: Although `zsh` and `bash` sometimes require spaces and sometimes forbid them, they're optional in mathematical expressions. So, `number=$((7*5+3))` and `number=$(( 7 * 5 + 3 ))` both work.

Learn More about Shell Scripting

You can find oodles of websites dedicated to teaching scripting in `zsh` and `bash`—from beginner to advanced levels. Here are a few examples:

- Apple's Shell Scripting Primer
- Dealing with basic shell syntax (for zsh)
- Shell Scripting Tutorial

Using Terminal in Recovery Mode

If your Mac has issues that keep it from booting properly, you might use recovery mode to run Disk Utility or perform other maintenance. While in recovery mode, you can also use Terminal, which can come in handy for running certain commands—for instance, finding files on your Mac and copying them onto a flash drive.

In particular, you'll need Terminal to reset a forgotten administrator password. Here's how you do it:

1. Restart your Mac in recovery mode (see About Recovery Mode).
2. Choose Utilities > Terminal. A Terminal window opens, running the `bash` shell (yes, even in Monterey).
3. Type `resetpassword` and press Return. You may have to wait a moment or two, but a new window called Reset Password opens. If it's behind the Terminal window, click it to bring it to the front.
4. Select your startup volume. From the Select the User Account pop-up menu, choose your username.
5. Enter and confirm a new password. Click Save, and then click OK to confirm the password reset.
6. Choose Reset Password > Quit Reset Password; then choose Terminal > Quit Terminal.
7. Choose macOS Utilities > Quit macOS Utilities and click Restart.

Note that in recovery mode, the `bash` shell offers only a subset of its regular commands (and the only other shells available are `sh` and `dash`—`zsh` isn't present). To see what commands are available, enter `ls /bin /sbin /usr/bin /usr/sbin`. Or press Esc twice, as discussed in the sidebar Which Programs Can I Run?.

Install New Software

With just the software macOS includes (and perhaps a few shell scripts you write on your own or find on the web), you can do a tremendous number of useful activities on the command line. But sooner or later you're likely to encounter a task that requires a command-line program you don't already have, which means you'll have to find and install it yourself. (Admittedly, this is not for everyone, and if the next few paragraphs give you a headache, skip ahead to Command-Line Recipes and forget I ever mentioned installing your own software!)

Fortunately, the vast majority of command-line software created for Unix and Unix-like operating systems (such as the various Linux distributions) can run on your Mac too! (Refer back to What's Unix? for the differences between "Unix" and "Unix-like.") Tens of thousands of command-line programs are at your disposal! Just a handful of examples:

- **alpine:** An email client
- **ffmpeg:** A tool for recording, converting, and streaming audio and video
- **lynx:** A command-line web browser (yes, really)
- **pdftohtml:** A program that converts—you'll never guess!—PDFs to HTML format
- **postgresql:** A relational database manager
- **wget:** A tool for downloading files from the web

Except… on the command line, it's almost never as simple as downloading an app and running it. Because each Unix and Unix-like operating system is a bit different, in most cases, a given program must be *compiled* for the specific platform in question—that is, the raw source code (in a language such as C) has to be processed by a program called a *compiler* to produce a *binary* file that runs on the target system. (In fact, compiling can be vastly more complex than this description suggests, but that's the basic idea.)

So, if you have an interest in adding third-party command-line software to your Mac, you'll first need the tools that are required to compile and install them. You can get them easily (read Use Command Line Tools for Xcode, next), and in the process gain a bunch of extra programs that may be useful to you on their own.

Next, you have a choice:

- If you're a glutton for punishment (or want to see how things work), you can Install Unix Software from Scratch. (Do it at least once in your life, just for the experience.)
- If you'd like to make life easier for yourself, however, you can often use a special program called a *package manager* to do the heavy lifting of finding, downloading, and (if necessary) compiling the software you want (see Use a Package Manager). Package managers are way faster and more convenient than compiling software from scratch, although not every program you may want to install is available in that form.

Use Command Line Tools for Xcode

Let's start with something simple: a free software package from Apple called Command Line Tools for Xcode. This collection includes nearly 100 new command-line programs, mostly intended to perform functions useful to developers but not needed by the typical Mac user. However, since you now know your way around the command line, you're not a typical Mac user! And in order to install new command-line software, you'll almost certainly need tools such as `make` (to build a set of binary files from their source files), which in turn relies on a compiler such as `gcc`.

Both of these programs and dozens of other development tools are in this set, as well as such goodies as:

- **GetFileInfo:** A command-line program that does something similar to the Finder's Get Info window
- **git:** The git version control system
- **svn:** The Subversion version control system

You can obtain and use these command-line tools with or without Xcode, Apple's software development system. Xcode is a free download from the Mac App Store, but it's nearly 8 GB in size and takes up much more space than that after it's installed. If you already have Xcode on your Mac, you can add the Command Line Tools with this command:

```
xcode-select --install
```

Follow the prompts to complete the installation.

If you don't have Xcode and don't want to bother with it, you can download the Command Line Tools separately (less than 200 MB). The catch is that you have to enroll in the Apple Developer Program—but you don't necessarily need to join the *paid* program for Mac developers; you can register for free to get access to Xcode and other tools.

Once you've done that, go to Downloads for Apple Developers, sign in with your Apple ID, and then download the version of Command Line Tools that corresponds to your version of macOS. Double-click the installer and follow the instructions.

After you've installed the Command Line Tools for Xcode, you can immediately run any of the commands it includes. You can also install software from other sources, as covered in the remainder of this chapter.

Note: Xcode developer tools were previously stored in their own directory, making it easy to see which items were new. Now they're added to the existing list of items in `/usr/bin`. If you have the full Xcode app, you can see what goes into this folder by right-clicking (or Control-clicking) the app, choosing Show Package Contents from the contextual menu, and navigating to `Contents/Developer/usr/bin`.

Install Unix Software from Scratch

Let's suppose you're looking for a command-line program that does X, and sure enough, you run across a webpage with what appears to be exactly the thing you want, a program I'll call `abc`. But what the site offers is not a compiled binary for macOS—it's just a bunch of source files, so you have to build and install it yourself. How do you proceed? Although the procedure can vary greatly, I want to show you the basic steps involved in a typical installation.

But first, let me give you two key pieces of advice:

- Before you do anything else, check to see if the software is available via a package manager (such as Homebrew, MacPorts, or Fink, discussed ahead in Use a Package Manager)—this is often noted on webpages where you can download Unix software. If so, installing the package manager, and then using that to install the `abc` program, is almost certainly the path of least resistance. I'd especially recommend using a package manager if you plan to install a different version of something that's included with macOS, such as PHP or Apache, because compiling your own and installing it manually could lead to unexpected conflicts.

- Look for installation instructions. In the vast majority of cases, the developer lists the exact steps to follow (sometimes, even including the download step), and if there are any variations for particular operating systems, they're often included in these instructions. When in doubt, do exactly what the developer says.

Having read and followed many such instructions myself, I can tell you that they usually involve this sequence: download, configure, make, and make install. I explain those (and a couple of additional important steps) next.

Download

If you're using a macOS web browser to locate the software you want to install, you can click a link to download it just as you would any other file. Once you've done that, you might want to move the downloaded file out of your Downloads folder to somewhere more convenient, but that's up to you.

On the other hand, if you already have a Terminal window open, you can download software directly to your current directory, using the `curl` command and the URL. (If you don't see the URL but just a link, you can right-click (Control-click) the link and choose Copy Link to put the URL on your clipboard.) To download the file, type `curl -O` (that's an uppercase o, not a zero) followed by a space and the URL, as in:

```
curl -O http://some-website.com/something/abc-1.2.3.tgz
```

In this example (as very often occurs), the file that downloads includes the name of the program (`abc`) and a version number (`1.2.3`).

Tip: For more on using `curl`, see the recipe Download a File.

Decompress

Because command-line software often includes many source files that must be compiled to make the final product, they're typically archived into a single file (often using a program called `tar`, for "tape archive") and then compressed (often using a program called `gzip`). The resulting file usually ends in `.tgz` or `.tar.gz`. (I hasten to point out that there are many other ways to archive and compress files, and thus many other extensions in use; this is just an example.)

If you've downloaded the file using a macOS web browser such as Safari, it may be decompressed automatically, at which point you'll end up with a folder (such as `abc-1.2.3`) in your Downloads folder.

If not, open a Terminal window, navigate to the folder containing the downloaded file, and enter (substitute the actual filename, of course):

```
tar -zxvf abc-1.2.3.tgz
```

If the file ends in `.bz2`, use this instead:

```
tar -jxvf abc-1.2.3.tar.bz2
```

At this point you'll have a folder (such as `abc-1.2.3`) containing the files you need to work with.

Read "Read Me"

Now stop for a moment. Look through the files in that folder (either in the Finder or on the command line, using the tools you've already learned about in this book, such as `cd` and `ls`). You will very likely find one or more files with names like `README` or `INSTALL`. These contain information about the program (`README`) and how to install it (`INSTALL`). They're invariably plain text files that you can open in a text editor (TextEdit, BBEdit, `nano`, or whatever) or view using a program such as `less` or `cat`. In any case, *read them*. They'll contain important instructions, and whatever they say takes precedence over what I tell you here!

One of the important things you might discover in a `README` file (or on the web) is that the software you're trying to install has certain *dependencies*—that it, it could rely on another program (or a *library*, which supplies features that any program can tap into) which must already be installed before the program will work. And that dependency might, in turn, have other dependencies. Working through those can sometimes be a long and frustrating process, which a reason to consider using a package manager when possible (see Use a Package Manager).

Configure

One of the instructions in the `README` or `INSTALL` file should say whether you need to perform a configuration step. This isn't always necessary, and when it is, sometimes the preferred method is to edit a text file with information about your system. But more often than not, the step you take at this point is to run a script called `configure`. Assuming you're in the same directory as the configure script, you do it like this:

```
./configure
```

The job of the `configure` script is to create a file called a *makefile,* which in turn contains all the instructions needed to compile the program for your particular computer. In most cases, `configure` doesn't require any interaction; you just run it and move on to the next step.

Make

So, that `makefile` you just made in the last step with the `configure` command? Here's where you use it. Assuming once again that you're in the directory where the software resides, simply enter:

```
make
```

That's it. The make command follows the instructions in the `makefile` to compile binary files for your Mac from the source files provided. This process may take anywhere from less than a second to many minutes or more, depending on the complexity of the software. You'll probably see messages in Terminal as the build progresses. You'll know the process is done when you see your command prompt again.

Make Install

Like macOS apps, command line programs sometimes require lots of components to exist in specific places, beyond the executable file itself. Now that you have created all those components with the `make` command, it's time to put them in the right locations and assign the proper permissions. To do so, enter:

```
sudo make install
```

Even for large, complex installations, the `make install` command is usually quite speedy. Once it has finished, you can run your newly installed software just as you would any other command line program.

Use a Package Manager

Now that you know the manual way to install command-line software, let's look at a simpler approach: using a type of software known as a *package manager*. This whole rigamarole of figuring out what dependencies a given program has; downloading, configuring, making, and installing all of them; and then downloading, configuring, making, and installing the program you want, can all be automated into a single-step process. That's what package managers do—they handle all the tedious details for you.

Although each package manager has its own methodology, in most cases, package managers download and install prebuilt binaries of the software you're interested in (as well as any dependencies), which saves time, disk space, and hassle. If a binary isn't available, if the latest available binary is out of date, or if there's some complicated reason why it's better to compile a particular program on your own Mac, the package manager can still do all that for you.

Not every command-line program you could want is available via a package manager, and of those that are, some are available only in certain package managers but not others. Nevertheless, many thousands of command-line tools can be installed using one package manager or another, including all the most popular tools and programs.

Tip: You will almost certainly have to install the Command Line Tools for Xcode before installing or using a package manager, as most package managers (including Homebrew, MacPorts, and Fink) require one or more or those tools. See Use Command Line Tools for Xcode, earlier.

I'm aware of at least six reasonably full-featured package managers for macOS, of which three (Homebrew, MacPorts, and Fink) are distinctly more popular than the others (Pkgsrc, Nix, and Rudix). And, of the three "cool kids," almost anyone will tell you that the real contest these days is between the venerable MacPorts and relative newcomer Homebrew. I’ll say a bit about each package manager, but to some extent, you can’t make a bad choice; as long as the one you use offers the package that you’re interested in, it’ll be way easier than starting from scratch.

As you choose a package manager, keep these tips in mind:

- Pay attention to where on your disk the package manager stores binaries, and whether you have a choice in the matter. There are good reasons to choose any of several locations, but some of them are controversial (I’ll give an example when I talk about Homebrew).
- Whichever location your package manager uses for binaries, it must be included in your PATH for the software to operate correctly. That’s one advantage of Homebrew’s use of `/usr/local/bin`—that’s already in your PATH by default. To make sure the binary location is in your PATH, follow the steps in Modify Your PATH.
- Package managers differ in how they treat dependencies. Some try to supply all their own dependencies, while others rely as much as possible on programs and libraries included with macOS. The former approach can take longer, use more space, and leave you with duplicates of certain programs. But the latter approach could break your programs when Apple updates macOS and in so doing removes a dependency (or supplies an incompatible version). There’s no right answer here, just different approaches to weigh.
- Under some circumstances, it might be possible to use more than one package manager at the same time, but I recommend against it. If you should happen to install the same software with each of two package managers, it’ll be hard to predict which one runs when you enter the program’s name (it’s the one whose path happens to be listed first in your PATH), and dependencies could get complicated.

With that background, here’s an overview of Mac package managers. (To download and install any of them, follow the instructions provided on their respective websites.)

Tip: If you don’t want to study all the tedious differences among the package managers and just want to *pick one*, start with Homebrew, which I cover next. If you find you need a package it doesn’t offer, you can then move on to MacPorts and so on—the remaining package managers are listed in order of my personal preference.

Note: One package manager I omit here is Conda, which specializes in Python programs for data scientists (even though the framework can be used with other languages and program types). I also found a package manager called Zero Install that claims to work on Mac, Windows, and Linux, with over 2,000 packages available, but I was unable to install it on Mojave or later, which suggests that its Mac support is inadequate at best.

Homebrew

Homebrew is arguably the most modern, straightforward, and easy-to-use package manager—it has a lot less baggage and clutter than MacPorts and Fink, and it works on Linux as well as macOS. On the other hand, it also has fewer packages than its main competitors—over 7,000 as of early 2024—though that number includes many of the most widely used programs, and many useful tools that are unavailable in other package managers. Speaking of which, Homebrew doesn’t use the term “packages”; instead, it’s riddled with beer-brewing metaphors. For example, a given program is offered either as a *formula* (instructions to download and compile the software) or as a *bottle* (a compiled binary).

Homebrew is written in Ruby, and specializes in tools of use to Ruby on Rails developers. It relies on existing macOS software when possible, making it less complex than Fink or MacPorts, but with a greater likelihood of something breaking after upgrading macOS. It does not use `sudo` for any of its work, making it less risky to use than other package managers.

Unless you expressly specify a location, Homebrew installs itself in `/opt/homebrew` on M-series Macs and `/usr/local` directory on Intel

Macs and; it uses `/opt/homebrew/Cellar` (M-series) or `/usr/local/Cellar` (Intel) for the binaries it installs.

Some usage examples:

- Show all packages Homebrew can install:
  `brew search`
- See if a particular program (`lynx` in this example) is available:
  `brew search lynx`
- Install the `lynx` package:
  `brew install lynx`

Tip: If you want to use Homebrew to install GUI Mac apps from the command line, a package called Homebrew Cask does just that. Conversely, if you want to use a GUI Mac app to install packages using Homebrew, you can use an app called Cakebrew.

Homebrew has become my go-to package manager, and it's probably the best one to try first, because the learning curve is so gentle. (And, as I said, it's great for Ruby on Rails developers.) For anything unavailable via Homebrew, my top choice would be our next contender: MacPorts.

MacPorts

MacPorts, written in the Tcl scripting language, started life in 2002 as DarwinPorts, and is based on the Ports system for BSD (which is appropriate since the Mac's Unix layer is itself based on BSD). It now has the largest selection of packages (called *ports*) available—over 38,000. MacPorts uses the `/opt/local` directory by default. Unlike Fink (covered next), it relies as much as possible on programs and libraries already installed as part of macOS.

Here are examples of how you might use MacPorts:

- Show all packages MacPorts can install:
  `port list`

- See if a particular program (`lynx` in this example) is available:
  `port search lynx`
- Update MacPorts' listing of available packages:
  `sudo port -d selfupdate`
- Install the `lynx` package:
  `sudo port install lynx`

Although MacPorts is not the easiest package manager to use (the documentation goes on forever), it's solid and has a thorough library. If you need its greater breadth of packages, MacPorts could be a great choice for you.

Fink

Fink is the oldest package manager for the Mac, having first appeared in 2000. It's written in Perl and based on a package manager for Debian Linux called `apt-get`. In order to use it, you must not only install the Xcode Command Line tools but also build Fink itself from source, as there's no binary installer for macOS versions higher than 10.5. As of February 2024 Fink had well over 20,000 packages, although many of those are outdated and no longer maintained. Fink tends to install its own dependencies rather than relying on software that's included with macOS. Prior to Catalina, Fink created and used the directory `/sw` by default, but since that requires writing to a location that is on the read-only system volume in Catalina and later (see The Startup Volume Became Read-Only), Fink now uses `/opt/sw` instead.

The Fink syntax should look familiar by now:

- Show all packages Fink can install:
  `fink list`
- See if a particular program (`lynx` in this example) is available:
  `fink list lynx`
- Update Fink's listing of available packages:
  `sudo apt-get update`

- Install the `lynx` package:
  `sudo apt-get install lynx`

Pkgsrc/pkgin

The Pkgsrc package management system uses a tool called pkgin for managing and installing binary packages, and the two terms are somewhat interchangeable, depending on the context. Unlike most of the other package managers listed here, pkgsrc (and pkgin) works on virtually every Unix and Unix-like operating system. (It was also written in C, making it immune to the eventual disappearance of any scripting languages from macOS.) As such, it might be a good choice if you also use it on another platform, but it's less tailored to the specific needs and preferences of Mac users. Pkgsrc defaults to using either the `/usr/pkg` or the `~/pkg` directory, depending on which installation mode you use. It currently offers over 19,000 binary packages.

Some syntax examples:

- Show all packages pkgsrc can install:
  `pkgin avail | wc -l`
- See if a particular program (`lynx` in this example) is available:
  `pkgin search lynx`
- Install the `lynx` package:
  `sudo pkgin -y install lynx`

Nix

Nix is a package manager for Unix and Unix-like operating systems (including, of course, macOS and Linux). Of the more than 80,000 packages available altogether, less than half can be used on a Mac. That is an impressive number, although many of those packages are simply older versions of given programs. Nix is written in C++, which means it has no dependencies on scripting languages.

In terms of user-friendliness, Nix is a far cry from the simplicity of Homebrew; it's clearly oriented mainly toward developers and other highly technical users.

Some syntax examples:

- Show all packages nix can install:
  `nix-env -qa`
- See if a particular program (`lynx` in this example) is available:
  `nix-env -qa lynx`
- Install the `lynx` package:
  `nix-env -i lynx`

Rudix

Rudix is written in Python, so it may be particularly attractive to Python developers; the usual disclaimers about the eventual disappearance of scripting languages from macOS apply. It has the smallest selection of packages by far—just over 300—but of course if that selection includes all the ones you care about, that's not an issue. Rudix offers self-contained packages with all dependencies included, except those provided by macOS libraries. It uses the `/usr/local` directory by default, much like Homebrew, but at least it doesn't change the ownership of that directory. On the downside, that means you'll have to use `sudo` to run the software Rudix installs.

Here are some example commands:

- Show all packages Rudix can install:
  `rudix search`
- See if a particular program (`lynx` in this example) is available:
  `rudix search lynx`
- Install the `lynx` package:
  `sudo rudix install lynx`

Learn Command-Line Shortcuts

Throughout this book I've mentioned tips, tricks, and shortcuts for performing a wide variety of command-line tasks, and in the next chapter (Command-Line Recipes) I show you dozens of prebuilt, ready-to-use commands and scripts. In this chapter, however, I wanted to pull together some of my favorite tips for using the Terminal app itself, and `zsh` in particular, that didn't quite fit anywhere else. As you become more adept at the command line, you're also likely to be more annoyed at inefficiencies, and this chapter aims to eliminate quite a few of them!

Terminal Tips and Shortcuts

I want to begin with a few words about useful but oft-overlooked features in the Terminal app itself—features that work regardless of which shell is running.

Note: All these shortcuts work in 10.11 El Capitan and later; many of them also work in earlier versions of macOS.

Paste Selected Text

Let's say you select some text in your Terminal window and you want to put that on the command line. You could copy it (⌘-C) and then paste it (⌘-V), but that's two steps, and you would lose whatever was previously on your clipboard. Terminal has a one-step command to do both together—without losing whatever else might be on your clipboard: with text selected, just press ⌘-Shift-V (or choose Edit > Paste Selection).

Select Output from the Previous Command

Many commands in Terminal produce output (think `ls`, `cat`, `locate`, and so on). If you want to select all and only the output from the most recent command (for example, to paste into a book you're writing about the command line!), press ⌘-Shift-A.

Erase Output from the Previous Command

Just as you can *select* output from the previous command (as above), you can *erase* that output—just press ⌘-L. (This erases the command itself from your Terminal window as well.) You can keep pressing ⌘-L to erase the output of previous commands, too!

Scroll Back to the Previous Command

If you want to *repeat* a previous command, you can use the arrow keys to place previous commands on the command line. But what if you don't want to repeat a command—you just want to scroll up in the Terminal window to where the last command was? Press ⌘-↑. That scrolls the display up to the previous command and highlights it. (You can also, of course, press ⌘-↓ to move to the next command.)

Tip: You can combine the last two items! For example, if you press ⌘-↑ followed by ⌘-Shift-A, you can select the output from *earlier* commands—not just the most recent one.

Search Your Command History

Yet another way to see (and reenter) commands you issued previously is to *search* them. Press Control-R to enter history search mode. The command prompt then looks like this (in `zsh`):

```
bck-i-search: _
```

In `bash`, it looks like this:

```
(reverse-i-search)`': _
```

Note: This command makes use of the backtick (`` ` ``) character, which is called a grave accent when placed over a vowel. It's on the same key as the tilde (`~`), and should not be confused with the apostrophe (`'`) or the backslash (`\`).

Now type a few characters that you remember being in an earlier command. If the right command appears, you can press Return to enter it again, if not, you can continue pressing Control-R repeatedly to look for additional matches.

Move the Cursor Left or Right by Word

As in most macOS apps, if you want to move your cursor a long distance on a line, you need not go character by character with the arrow keys. Press Option-← or Option-→ move the cursor back or forward one word at a time.

Zsh Tips and Shortcuts

If you want to learn every last detail about `zsh`, a good place to start would be its documentation, which is available online in HTML and PDF formats (among others). Now, the PDF is 544 pages long, so I hope you'll understand why I don't cover *every last feature* in this book! To be sure, a lot of the great things `zsh` can do are interesting only to programmers or hard-core shell users, and that's not the audience I wrote this book for. I do, however, want to point out a handful of the features I find particularly helpful or noteworthy, especially some that are not present in `bash` (and therefore new to many Mac users).

Navigate More Easily

One category of tasks in which `zsh` provides numerous improvements over `bash` is navigation from one directory to another. Here are some examples.

Use Substitutions to Change Directories

Say you're in /Applications/Xcode.app/Contents/Resources and you want to go to /Applications/Safari.app/Contents/Resources. Instead of retyping the whole pathname, you can do what amounts to an inline find-and-replace, with `cd old new`, replacing *old* with the part of the current path you want to replace and *new* with the corresponding part of the new path:

```
cd Xcode Safari
```

That reads: change to the directory that's just like the one I'm currently in, except replace Xcode with Safari in the path. Presto! You're now in /Applications/Safari.app/Contents/Resources.

Change Directories Without cd

It's not exactly onerous to type cd and a space to change directories, but if you're anything like me, you occasionally forget the cd and get confused when you don't find yourself in the directory whose name you just entered. In zsh, you can add a line to your .zshrc file that eliminates this problem:

```
setopt autocd autopushd pushdignoredups
```

As usual, this change takes effect when you start your next shell session or enter source ~/.zshrc.

With that in place, if you want to jump to /usr/local/bin, just enter:

```
/usr/local/bin
```

No cd required!

List Recent Directories

As you work in zsh, you may visit lots of directories with long pathnames, and it can be a drag to keep cycling through recent commands to move to a specific directory. There's an easier way—just enter:

```
dirs -v
```

When you do, zsh spits out a numbered list of up to 10 recently visited directories. If you find one you want to go back to, type a tilde (~) followed by that directory's number in the list—for example:

```
~3
```

Let zsh Correct Your Mistakes

We're all used to spelling checkers in word processors, email apps, and the like, but since the command line is particularly sensitive to having everything spelled exactly right, wouldn't it be nice if your shell could correct simple typos for you? If your shell is zsh, it can!

To enable this feature, first edit .zshrc to include the following line:

```
setopt correct
```

Start a new shell session or enter source .zshrc. Now try some minor misspellings. For example, suppose you mistyped pwd as pws. When you press Return, zsh offers to correct it:

```
% pws
```

```
zsh: correct 'pws' to 'pwd' [nyae]?
```

At this point, you can type y for yes, n for no (meaning zsh will try the command just as you entered it, even if it fails), a for abort (to cancel the command), or e for edit (which repeats the command, putting the cursor at the end of the misspelled word so you can correct it yourself).

Open Files Automatically

We've talked about aliases already (see Create Aliases), but zsh has a couple of extra alias tricks. One of them called suffix command aliases, lets you open a file in a predetermined program, based on its extension, simply by typing its pathname. (This may sound suspiciously similar to Change Directories Without cd, and that's not a coincidence!)

For example, suppose there's a file in the current directory called file.txt. Normally, to open that in nano, you'd enter nano file.txt.

But if you *always* want to use nano to open text files, you can add an alias line to `.zshrc` that specifies a combination of extension(s) and program name. For example:

```
alias -s {txt,csv,sh}=nano
```

Once you've started a new shell session (or entered `source ~/.zshrc`), all you have to do to open `file.txt` (or `list.csv` or `script.sh`, in this example) is enter the filename—for example:

```
file.txt
```

You can, of course, include whichever extensions inside the braces you want, and you can add more `alias -s` lines for other programs.

Use Global (Inline) Aliases

Speaking of aliases…so far, we've discussed two types—the regular kind (in which an entire command, like `ls`, is replaced with a different command, like `ls -lah`), and suffix command aliases (just above), which let you open files in a specified program based on their extension. But `zsh` has yet another type of alias: *global* (also known as *inline*) aliases. A global alias is one you can include anywhere on the command line, even if there are other things before and/or after it.

For example, consider the following command, from back in Get a Grip on grep:

```
ls /Library/Preferences | grep -ERI "Apple"
```

Here we're piping the output of `ls` into `grep` to show just a subset of the listing. If that's the sort of thing you do frequently—perhaps with different terms—you could make an alias to *just part of that command*. For example, if you put this in `.zshrc`:

```
alias -g GR="| grep "
```

Then, once you start a new shell session or enter `source .zshrc`, anytime you type `GR` on the command line, it'll be replaced with `| grep` (and a trailing space). To wit:

```
ls /usr/bin GR uu
```

On my Mac, that outputs:

```
-r-xr-xr-x   1 _uucp  wheel  139K Feb  3 01:10 cu
-r-xr-xr-x   1 _uucp  wheel  108K Feb  3 01:10 uucp
-r-xr-xr-x   1 root   wheel   37K Feb  3 01:10 uudecode
-r-xr-xr-x   1 root   wheel   31K Feb  3 01:09 uuencode
-rwxr-xr-x   1 root   wheel   67K Feb  3 01:10 uuidgen
-rwxr-xr-x   1 root   wheel   77K Feb  3 01:10 uulog
-r-xr-xr-x   1 _uucp  wheel   60K Feb  3 01:10 uuname
-rwxr-xr-x   1 root   wheel   87K Feb  3 01:10 uupick
-r-xr-xr-x   1 _uucp  wheel  117K Feb  3 01:10 uustat
-rwxr-xr-x   1 root   wheel  1.9K Nov  9 02:39 uuto
-r-xr-xr-x   1 _uucp  wheel  108K Feb  3 01:10 uux
```

That is, it lists all the lines that include the string “uu” anywhere. It’s equivalent to:

```
ls /usr/bin | grep  uu
```

Any segment of a command you use regularly can be aliased this way.

Learn Handy Keyboard Shortcuts

We’ve already covered quite a few keyboard shortcuts throughout this book, but I wanted to add a few more that work in `zsh` (and some other shells) and help you edit whatever’s on the current line:

- **Control-A:** Move the cursor to the beginning of the line.
- **Control-E:** Move the cursor to the end of the line.
- **Control-K:** Delete everything from the cursor to the end of the line.
- **Control-W:** Delete everything from the cursor backwards to the start of the preceding word.
- **Control-U:** Delete everything on the line.

Use Oh My Zsh

Oh My Zsh is an optional framework you can add to `zsh` that enables you to load user-contributed themes (which change the prompt in various useful and/or entertaining ways) and plugins (shell scripts that give `zsh` additional features, most notably autocompletion for various languages and platforms).

Some `zsh` users think Oh My Zsh is the Best Thing Ever and would never use `zsh` without it; for others (including me), it's sort of *meh*. But that's because I like to make my own customizations just so, and because I don't do much command-line work in the ecosystems (such as Git, Python, and Ruby) where Oh My Zsh is most effective. Nevertheless, I'd be remiss if I didn't tell you about it and encourage you try it out!

You can install it with this command:

```
sh -c "$(curl -fsSL https://raw.github.com/ohmyzsh/ohmyzsh/master/
tools/install.sh)"
```

Note: You may see a warning message about permissions for certain files. If you do, to fix it, enter the following, substituting your short username for *user*: `sudo chown -R user /usr/local/share/zsh/site-function`

You'll notice that your prompt changes dramatically, and any customizations you'd previously put in `.zshrc` disappear. That's because the Oh My Zsh installer uses its own custom `.zshrc` file. (But don't worry, it keeps a backup copy of your old file at `~/.zshrc.pre-oh-my-zsh`; you can manually copy stuff from there into your new `.zshrc` file if you like, and if you uninstall Oh My Zsh using `uninstall_oh_my_zsh`, the uninstaller puts back all your old settings.)

Now that you're running Oh My Zsh, you can:

- **Change your theme.** To do this, edit `.zshrc` and look for the line near the top that reads:

  ```
  ZSH_THEME="robbyrussell"
  ```

Replace *robbyrussell* with the name of the theme you want to use—you can see your options on this page.

- **Enable a plugin.** To do this, edit `.zshrc` and scroll down near the end of the file, where you'll see the line:

  ```
  plugins=(git)
  ```

  The parentheses should contain the full list of plugins you want to load, separated by spaces—for example:

  ```
  plugins=(git brew emacs lol)
  ```

  You can find a list of available plugins on this page.

Command-Line Recipes

You've learned about the raw ingredients and the tools you need to put them together. Now it's time for some tasty recipes that put your knowledge to good use! In this chapter, I present a selection of short, easy-to-use commands and customizations you can perform in the `zsh` or `bash` shells. Many use features, functions, and programs I haven't yet mentioned, and although I describe essentially how they work, I don't go into detail about every new item included in the recipes. Just add these herbs and spices as directed, and enjoy the results!

Misplaced hyphens! Your ebook reader may insert extra hyphens into longer lines of command-line text shown in this ebook. Please see the first item under Basics, earlier, for more information about how to avoid extra hyphens.

Change Defaults

Most macOS apps, from the Finder to Mail to Music, store their settings in specially formatted property list, or `.plist`, files. Surprisingly, apps often have hidden preferences that don't show up in their user interfaces—but if you put just the right thing in the preference file, you can change an app's behavior in interesting ways, or even turn on entirely new features.

One way to edit preference files is to open them in a text editor, or in Apple's Xcode development environment (which is available as a free download from the Mac App Store). But another, often easier way, is to use a command called `defaults` which can directly add, modify, or remove a preference from a `.plist` file. The recipes in this first set all use the `defaults` command.

Before using these commands to alter an app's defaults, quit the app if possible (obviously that's not an option with the Finder or the Dock, but the recipes that involve those apps include directions to force-quit and relaunch them).

Tip: Various websites list hundreds of additional settings you can change—for example,see defaults-write.com and dotfiles.

Show Hidden Files in the Finder

By default, the Finder keeps some files and folders hidden—those whose names begin with a period and many of the Unix files and directories at the root level of your disk.

That's usually good, because it prevents you from changing things that could cause your computer to break, but if you want to see all your files and folders, enter this:

```
defaults write com.apple.finder AppleShowAllFiles true; killall Finder
```

(To reverse this setting, repeat the command, changing `true` to `false`.)

Change the Screenshot Format

When you take a screenshot in macOS (using the ⌘-Shift-3, ⌘-Shift-4, or ⌘-Shift-5 keyboard shortcuts), the resulting pictures are normally saved, on your desktop, in PNG (Portable Network Graphics) format. If you prefer another format, such as JPEG, enter this:

```
defaults write com.apple.screencapture type -string "jpeg"; killall SystemUIServer
```

Use the same command, but substitute `bmp`, `gif`, `pdf`, `png`, or `tiff` for `jpeg` to use one of those formats.

Note: You may notice these file types are in lowercase. If you use uppercase instead, the command still works, but the file extension will also be uppercase. Conversely, if you're currently getting screenshots with uppercase extensions and you prefer them to be lowercase, this command can solve that problem for you.

Use Single-App Mode

If you have lots of apps open and find all that screen clutter visually distracting, you could manually hide each app (other than the one you're currently using), or you can switch to an app while simultaneously hiding all the rest by Option-clicking the app's Dock icon.

But if you'd prefer to have macOS hide all background apps automatically, you can do so with this command:

```
defaults write com.apple.dock single-app -bool true; killall Dock
```

After you do this, switching to any app not only brings it to the front but also hides any other running apps (without quitting them). To return to the standard behavior, repeat the command but with `false` instead of `true`.

Show Half-Star Ratings in Music or iTunes

I don't know about you, but I have a lot of tracks in Music that feel to me like *not quite* five stars, but *a little better than* four stars. If you want to be able to apply half-star ratings in Music (or in iTunes, if you're using Mojave or earlier), you can do so with a quick command. First quit Music or iTunes and then enter the appropriate line:

- **Music:** `defaults write com.apple.Music allow-half-stars -bool true`
- **iTunes:** `defaults write com.apple.iTunes allow-half-stars -bool true`

Then reopen Music or iTunes. When you click a star icon to rate a track, you can move the pointer just slightly to the left or right to partially fill in that star. To go back to only full-star ratings, repeat the command, replacing the `true` at the end with `false`.

Disable Inline Attachment Viewing in Mail

If you receive or compose email in Apple Mail that includes graphical attachments (such as JPEG or PNG files), Mail displays the images right there in the body of the message. If you'd prefer all such images to be represented by icons (which you can then double-click to view the full image), quit Mail and enter this command:

```
defaults write com.apple.mail DisableInlineAttachmentViewing -bool
true
```

Reopen Mail, and you'll see only icons for attached images. To undo this command, repeat it, replacing `true` at the end with `false`.

Disable App Nap

In 10.9 Mavericks, Apple introduced a performance-enhancing, energy-saving feature called App Nap. App Nap intelligently reduces the system resources used by background apps, in order to make the foreground app more responsive while using less power overall. However, sometimes you may not want your background apps to nap—you may want them to have access to your full system resources, for example to perform a complex calculation or graphics operation while you work on something else.

Disabling App Nap globally can dramatically shorten your battery life, but if background performance is more important to you than power savings and you'd like to disable App Nap globally, you can do it like this:

```
defaults write NSGlobalDomain NSAppSleepDisabled -bool true
```

As usual, reenable it by replacing `true` with `false`.

Use a Separate Password for FileVault

Do you use FileVault to protect your Mac's data? Good for you! Ordinarily, you unlock FileVault in the process of logging in with your regular account password. But if you're extremely security-conscious and want to use a *different* password for FileVault than your login password, you can—using a `defaults write` command, of course. Be aware that you'll face two consecutive login prompts whenever you start or restart your Mac—the first to unlock FileVault, and the second to log in to your user account.

The way to accomplish this is to turn off FileVault's Auto-login feature, which normally logs you in to your account using the same password you just entered to unlock FileVault. To do this, enter:

```
sudo defaults write /Library/Preferences/com.apple.loginwindow
DisableFDEAutoLogin -bool true
```

After you do this, you'll get two password prompts when you restart, but the two passwords will still be the same. You can't (readily) change your FileVault password, but you *can* change your account's login password in System Settings/System Preferences > Users & Groups > Password by clicking Change Password and following the prompts.

To return to automatic login, first change your login password back to match your FileVault password, and then use this command:

```
sudo defaults delete /Library/Preferences/com.apple.loginwindow
DisableFDEAutoLogin
```

Remove the Proxy Icon Hover Delay in Big Sur

The proxy icon is the tiny icon at the top of a window next to the name of the document or folder. It's not just decorative; you can actually drag that icon to move the item, open it in a different app, or perform other actions on it. It serves as a handy shortcut, preventing you from digging around for the original file or folder icon.

In Big Sur and later, owing to the rearrangement of window tops, the title *bar* as such disappeared; the title now appears to the left of the main controls on the toolbar. Presumably as a matter of preserving horizontal space, the proxy icon is hidden by default—but it reappears when you hover your pointer over the title. Unfortunately, it doesn't appear *immediately*. You have to hover, and then wait a moment for the proxy icon to appear. If you use that icon frequently, as I do, that delay can become quite irritating.

Starting in macOS 12 Monterey, you can return the proxy icon permanently—without any delay—by going to System Settings > Accessibility > Display or System Preferences > Accessibility > Displays and enabling “Show window title icons.” In Big Sur, unfortunately, that option doesn’t exist. However, I can tell you how to do the next-best thing, which is to remove the *delay*, so that as soon as you hover over the title, the proxy icon appears!

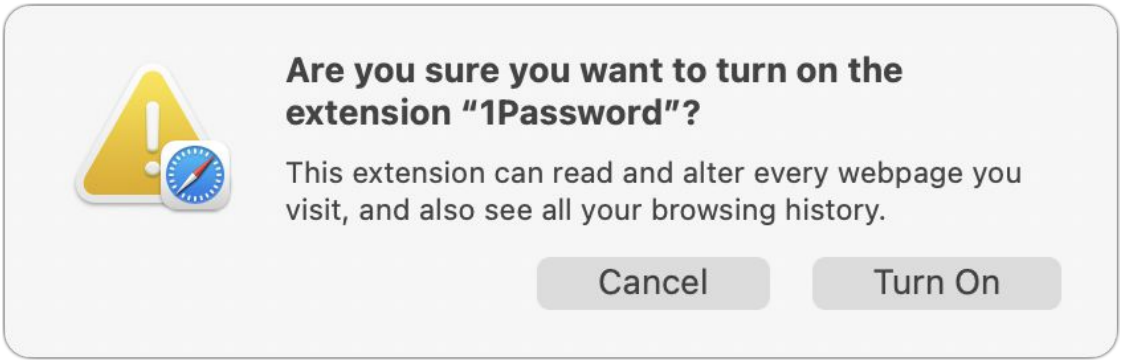
To do so, enter this command, which is actually two commands on the same line (the second of which relaunches the Finder):

```
defaults write com.apple.Finder NSToolbarTitleViewRolloverDelay
-float 0; killall Finder
```

To reverse this and restore the hover delay, repeat the command, replacing the `0` with a `1`.

Use Old-Style Alerts in Big Sur and Later

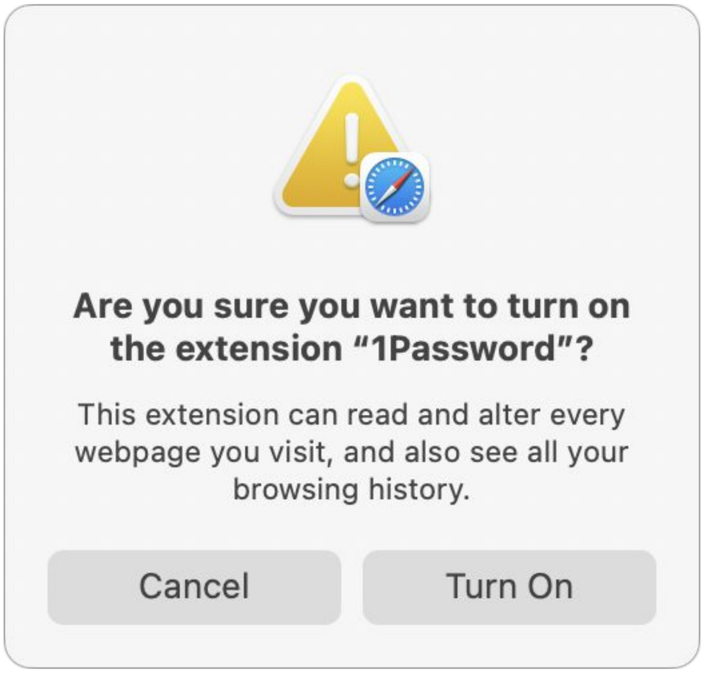
One of the controversial design changes starting in Big Sur was rearranging dialogs so that they’re oriented more vertically (as in iOS and iPadOS) than horizontally. You can see the difference in **Figure 10** and **Figure 13**. If you’re not a fan of the new style, you can change a preference that reverts many (though not all) dialogs to use the old style.



Are you sure you want to turn on the extension “1Password”?

This extension can read and alter every webpage you visit, and also see all your browsing history.

Figure 10: This is the old dialog style, with a horizontal orientation, used in Catalina and earlier.



Are you sure you want to turn on the extension "1Password"?

This extension can read and alter every webpage you visit, and also see all your browsing history.

Figure 11: And this is the new dialog style starting in Big Sur, which looks like what you see in iOS and iPadOS.

To use old-style dialogs (again, only in some places, depending on how developers designed their apps), enter this:

```
defaults write -g NSAlertMetricsGatheringEnabled -bool false
```

You'll have to quit and reopen any running app (or restart your Mac, as the case may be) to see the changes.

To revert to the default behavior, use this:

```
defaults delete -g NSAlertMetricsGatheringEnabled
```

Perform Administrative Actions

This group of recipes involves administrative tools—things you might need to do on a multi-user Mac, a server, or a remote Mac.

Use Software Update from the Command Line

Prior to Mojave, Apple used the App Store app for updating system software (as well as apps from both Apple and third parties). In Mojave and later, System Settings > General > Software Update or System Preferences > Software Update handles system updates, while the App Store is strictly for app updates. In any case, if you prefer to use the command line for updates that you'd otherwise use System Settings/ System Preferences or the App Store for—or if you want to update Apple software on a remote Mac via SSH—you can!

To update Apple system software (regardless of which version of macOS you're using), you can use the `softwareupdate` command. Enter this:

```
sudo softwareupdate -i -a
```

The `-i` and `-a` flags together mean "go ahead and install every available update."

What about the apps you'd normally update via the App Store app? For that, you'll need a third-party command-line tool called mas (available via Homebrew). The software's home page provides installation and usage details, but just as one example, you can use `mas upgrade` to install all pending updates.

List Your Mac's Reboot History

When did you last boot your Mac? A quick way to check is to enter:

```
last reboot
```

The answer will appear as a list in this format (possibly with earlier reboots listed afterward, depending on your Mac model and the version of macOS you're using):

```
reboot    ~                         Fri Feb  7 13:05
```

Show How Long Your Mac Has Been Running

A slightly different take on the previous recipe is a command that tells you the elapsed time since your last (re)boot (as opposed to the raw date and time):

```
uptime
```

The answer will look something like this:

```
19:09  up 4 days, 22:32, 3 users, load averages: 1.47 1.82 1.87
```

In this display, the first group of numbers is the current time. That's followed by how long the Mac has been running since its last (re)boot. In this example, it's been up for 4 days, 22 hours, and 32 minutes. The remainder of the line shows the number of users and load averages over the last 1, 5, and 15 minutes—all of which you can usually ignore.

List Users Who Logged In Recently

Is your Mac used by a number of different people? Do users sometimes log in remotely? If you'd like to know who's been logging in recently, you can get a lengthy list with this command:

```
last
```

This command lists the users who have logged into this machine (including both a normal macOS login and starting shell sessions), the IP address or terminal from which they logged in (if applicable), how long the user was logged in, and important system events such as shutdowns and restarts. This list may be very long!

Create and Edit User Accounts

Ordinarily, you'll use the Users & Groups pane of System Settings/ System Preferences to create and modify user accounts. But everything you do there can also be done on the command line, which might be the ideal environment if you have to automate the creation of user accounts (using a script) or create accounts on remote Macs via SSH.

The tool you want is `sysadminctl`, which is included with 10.13 High Sierra and later. For a detailed look at what `sysadminctl` can do, consult Using sysadminctl on macOS by Charles Edge. But here's a taste. To create a new user with a short name, full name, and password, enter:

```
sysadminctl -addUser shortname -fullName "Full Name" -password password
```

For example:

```
sysadminctl -addUser carol -fullName "Carol Peletier" -password
apocalypse
```

To change a user's password and add a hint, you use a similar command—for example:

```
sysadminctl -resetPasswordFor carol -newPassword Shiva -password-
Hint "The tiger's name"
```

Reset a Lost Password

Another activity normally performed in System Settings/System Preferences > Users & Groups is changing passwords. But what if you don't remember your own password—or the password for *any* administrator account on your Mac? Then Users & Groups is no use to you, but you're not out of luck.

Follow these steps:

1. Restart your Mac in recovery mode (see About Recovery Mode).
2. When the window appears, choose Utilities > Terminal.
3. In the Terminal window, type `resetpassword` and press Return.
4. Follow the prompts. Depending on which version of macOS you're running. whether FileVault is enabled, and whether you have a recovery key, you may see any of several different screens. One way or another, however, you'll be asked to enter and confirm a password and then restart your Mac.

Restart Automatically After a Freeze

If your Mac locks up completely while you're present, you can hold down the power button for several seconds until it turns off completely, and then press the button again to turn it back on. But if an unattended Mac freezes, it will sit there, frozen, until someone comes along to restart it. That could cause problems, especially when that Mac is functioning as a server, or if you need to access its files remotely.

To tell your Mac you want it to attempt an automatic restart in the case of a system freeze, enter this:

```
sudo systemsetup -setrestartfreeze on
```

This feature doesn't work all the time, but it's worth a try. Repeat the command with `off` instead of `on` to return to the default behavior of staying frozen until you do something about it manually.

Download Old Versions of macOS

If you're working with an older Mac, testing software, or troubleshooting, you may sometimes have to download versions of macOS older than what you're currently using, and the version you want may not be visible in the Mac App Store.

Although Apple offers direct links to installers for 10.10 Yosemite and later (see Redownload Archived macOS Installers to Address Expired Certificates in TidBITS), sometimes these links result in tiny "stub" installers that download the rest of their data as they run. You might reasonably prefer the full installer, especially if you have to reuse it repeatedly or if you're creating a bootable installer volume. Similarly, you might want a *specific* older version—for example, not just *10.14* but *10.14.2*.

If your Mac is running Catalina or later, you can use Terminal to download a full installer for a specific older version of macOS:

```
softwareupdate --fetch-full-installer --full-installer-version version
```

For example:

```
softwareupdate --fetch-full-installer --full-installer-version 10.14.2
```

The `softwareupdate` program downloads the installer you requested to your `/Applications` folder (but doesn't launch it). Although not every older version of macOS is available this way, it's always worth a try.

Find Interesting Stuff in Log Files

Many Unix and macOS apps and background processes constantly spit out log files detailing what they've been up to and, perhaps most important, any errors they've encountered. Most system processes store their log files in `/var/log`, and although you can open most of them in any text editor, they tend to be so long and inscrutable as to make the exercise more bother than it's worth. Luckily, you can use the `grep` command to sift through log files looking for specific strings.

For example, to look through the main system log for every instance of the word `error` (a sure sign of an interesting entry), enter this:

```
grep error /var/log/system.log
```

Or, to look for all entries involving Time Machine (whose background process is called `backupd`), enter this:

```
grep backupd /var/log/system.log
```

If you'd rather have a paged display instead of dumping the output directly onto the command line, you can pipe it through `less`, like so:

```
grep backupd /var/log/system.log | less
```

There is, however, a bit more to that story. In 10.12 Sierra, Apple moved to a so-called "unified" log format, which puts log entries from many apps in a single database, rather than writing them to separate text files. Although the above commands still work, the new log format makes it more challenging to find certain types of information. As much as I'd love to describe all the ins and outs of unified logging, I'd much rather spend an afternoon at the dentist. However, I can at least offer a couple of tips:

- In recent versions of macOS, if you open the Console utility and select your Mac's name in the sidebar, you get a real-time streaming list of new entries to the system log. Should you want to get the same dynamic view in Terminal, use this command:
  ```
  log stream
  ```

- Of course, those messages may go by too quickly to be useful, and perhaps you want to filter the display to show only some of them. In Console, you can type a term into the Search field to do so. On the command line, you can use this, where *string* is replaced with whatever you're looking for:

  ```
  log stream --predicate 'eventMessage contains "string"'
  ```

  If you want to filter by a specific process name, do it like this, where *process* is the name of the process you want (like `Safari`):

  ```
  log stream --predicate  '(process == "process")'
  ```

Here are a couple of links you can check out for a more detailed discussion:

- Inside Catalina's unified log: how has it changed? at the Eclectic Light Company
- Show private log messages in Catalina's Console.app by George Garside

Modify Files

A number of common recipes involve modifying files in some way. Here's a selection.

Change the Extension on All Files in a Folder

In Yosemite and later versions of macOS, the Finder has a built-in batch renaming function. But if you're using an older version of Mac OS X, or if you simply want a handy way to rename a large number of files in one go from the command line, this recipe is for you.

The `mv` command, discussed in Move or Rename a File or Directory, has an unfortunate shortcoming in that it can't rename a batch of files all at once with wildcards. But never fear, you can still get the job done.

Begin by creating the following shell script, using the instructions in Create Your Own Shell Script:

```
#!/bin/zsh
for f in $3/*.$1; do
   base=$(basename $f .$1)
   mv $f $3/$base.$2
done
```

Make sure it's located somewhere in your PATH, and that it's executable (see Understand Permission Basics, earlier). I call this script `chex.sh`, for "change extensions."

To run this script, enter the script name followed by the old extension, the new extension, and the directory in which to make the change—in that order.

For example, to change all the files in `~/Documents` with the extension `.JPG` to end in `.jpeg`, enter this:

```
chex.sh JPG jpeg ~/Documents
```

Decompress Files

If you decide to download Unix software (or source code to compile yourself), it may be packaged in any of several unfamiliar archive formats. As I mentioned in Decompress, a file ending in `.tar` is a "tape archive" (a way of bundling files together without necessarily compressing them); the extensions `.gz` and `.bz2` refer to different compression mechanisms, and you may see a combination of these (such as `archive.tar.gz`).

To decompress and/or unpack these, use one of the following commands, based on the extension(s) the file has:

```
tar -xf archive.tar
```

```
tar -xzf archive.tar.gz
```

```
tar -xjf archive.tar.bz2
```

As you can see, each compression format requires a different flag—use `-z` for `.gz` (or `.tgz`) and `-j` for `.bz2` (or `.bz`).

You may also, of course, encounter files compressed with Zip (`.zip`). You can decompress these in the Finder by double-clicking them, but if you want to do so on the command line, you can do it like this:

```
unzip archive.zip
```

Convert Documents to Other Formats

macOS includes a marvelous command-line tool called `textutil`, which can convert text documents between any of numerous common formats. This can be particularly useful in cases where you don't have Microsoft Word, or aren't satisfied with the way it saves files in other formats. Here are a couple of examples.

Convert a File from RTF to Word (.doc)

To convert the file `file1.rtf` (RTF format) to Word format (.doc) and save it as `file2.doc`, enter this:

```
textutil -convert doc file1.rtf -output file2.doc
```

Convert a File from Word (.doc) to HTML

To convert the file `file1.doc` (Word format) into HTML format and save it as `file1.html`, enter the following:

```
textutil -convert html file1.doc
```

(When no filename is specified, `textutil` uses the same filename with an extension corresponding to the format you requested.)

Note: The `textutil` program supports other formats too; just substitute the format of your choice for `doc` or `html` in the examples above. Among the most useful options are `txt` (plain text), html (HTML), `rtfd` (RTF with attachments), `docx` (Open Office XML), and `webarchive` (web archives, like Safari uses).

Manipulate Graphics

One of the terrific Unix gems included with macOS is a powerful program called `sips` (“scriptable image processing system”), which lets you manipulate graphics from the command line. In most cases, you’d want to adjust photos and drawings in a GUI app so you can see what you’re doing. But for simple, mechanical tasks like resizing, cropping, rotating, and flipping images, `sips` is an elegant solution. And, as its name implies, it’s especially well suited for shell scripts.

As usual, you can learn all about this program using `man sips`. But I wanted to show you a sampling of easy yet useful tasks you can use it for. All of these examples modify a file in the current directory called `image.jpeg`; obviously, substitute the path to whatever image you want to operate on.

Warning! As written below, these commands modify the target image rather than making a copy, and they can’t be undone. So, it’s always best to work on a copy (or make sure your script makes a copy first).

Resize an Image to a Fixed Height or Width

To make `image.jpeg` 400 pixels wide (keeping its height proportional to the original), enter:

```
sips --resampleWidth 400 image.jpeg
```

To make `image.jpeg` 400 pixels high (keeping its width proportional to the original), enter:

```
sips --resampleHeight 400 image.jpeg
```

To make `image.jpeg` 400 pixels in its longest dimension, regardless of which one that is (and keeping the other dimension proportional), enter:

```
sips --resampleHeightWidthMax 400 image.jpeg
```

Crop an Image to a Fixed Height and Width

To crop image.jpeg to 300 pixels high by 400 pixels wide (keeping the center point the same and discarding edges as necessary), enter:

```
sips --cropToHeightWidth 300 400 image.jpeg
```

Note: Although there is an additional parameter --cropOffset offsetY offsetH that can be added to --cropToHeightWidth to specify the origin point for cropping (in pixels from the top left corner), this parameter behaves unpredictably—especially when both offsetY and offsetH are set to 0. (Try it yourself and see: sips --cropToHeightWidth 300 400 --cropOffset 0 0 image.jpeg)

Rotate an Image

To rotate image.jpeg 90 degrees clockwise, enter:

```
sips --rotate 90 image.jpeg
```

Flip an Image

To flip image.jpeg about its horizontal axis, enter:

```
sips --flip horizontal image.jpeg
```

To flip image.jpeg about its vertical axis, enter:

```
sips --flip vertical image.jpeg
```

Tip: You can do much more with sips, including converting file formats. For some hints on how to do that (including writing scripts that can convert multiple files), see Converting Image File Formats with the Command Line & sips at OSXDaily.

Work with Information on the Web

The command-line environment includes a handy program called curl that can connect to web, FTP, and other servers and upload or download information. Here are a few examples of how to use it.

Download a File

If you know the exact URL of a remote file on a web, FTP, SFTP, or FTPS server, you can fetch it with this simple command (fill in the URL as appropriate):

```
curl -O URL
```

(Again, that's an uppercase letter o, not a zero.) The file is downloaded to your current directory.

Save a Local Copy of a Webpage

When you browse the web in Safari, you can save the source of any webpage. You can do the same right on the command line, without ever opening a browser, using this command:

```
curl URL > filename.html
```

For example, to save the source of the TidBITS home page to a file named `tidbits.html`, you can enter this:

```
curl https://tidbits.com/ > tidbits.html
```

Note that this command doesn't download graphics, style sheets, or other files linked from the webpage, so the page may not look entirely correct if you open it in a browser.

Put the Source of a Webpage on the Clipboard

What if you don't want to save a webpage to a file, but instead want to put it on your clipboard so you can paste it into another app? Enter the following:

```
curl URL | pbcopy
```

For example:

```
curl https://tidbits.com/ | pbcopy
```

Manage Network Activities

The following several recipes involve ways of getting information about local and remote networks, and the computers running on them.

Get Your Mac's Public IP Address

If your Mac is connected to the internet using a gateway, modem, or router, its private IP address (the one you can see in System Settings/System Preferences > Network) probably isn't the same as the public address that other computers see. (Depending on your ISP, your router model, and how your Mac and your router are configured, you may have an IPv4 address, like `12.34.56.78`, an IPv6 address, like `1234:5678:9000:abcd:ef01:2345:6789:abcd`, or both.)

To find out your Mac's current public IPv4 address, enter this:

```
curl -4 https://icanhazip.com/; echo
```

To see your Mac's current public IPv6 address, enter this:

```
curl -6 https://icanhazip.com/; echo
```

The `echo` command is there only to make sure there's a blank line after your IP address when it's reported, to improve readability.

Get a List of Nearby Wi-Fi Networks

The Wi-Fi 🛜 menu in your menu bar lists nearby Wi-Fi networks. But if you've turned off that menu, or simply want to get at that information from the command line, enter this:

```
/System/Library/PrivateFrameworks/Apple80211.framework/Versions/A/Resources/airport -s
```

Note: If you're reading this book in EPUB format and the above line is cut off, try zooming way out.

It displays nearby networks' names (SSIDs), MAC addresses, encryption types, and other useful information. To learn what else this tool can do, substitute the `-h` ("help") flag for `-s`. Yes, the full path is needed for executing this command: if you think you'll use it often, you can set up an alias for it (see Customize Your Defaults, earlier).

View Your Mac's Network Connections

Which servers and other network devices is your Mac currently connected to? For all the details (in fact, far more details than you probably want), try:

```
netstat
```

The `netstat` command spits out a tremendous amount of detail about which protocols are sending data to or receiving data from which addresses on which ports and a great deal more. It can be overwhelming but also fascinating to get a glimpse into what processes are doing various things online. (And don't forget, there's always `man netstat`—see Get Help.)

Find Out Which Apps Have Open TCP/IP Network Connections

You take it for granted that your web browser and email app are connected to the internet. But what other apps or background processes have network connections? Is anything covertly "phoning home?"

To see a list of processes you own that are accessing the internet right now, enter this:

```
lsof -i
```

To see a list of *all* processes accessing the internet, enter:

```
sudo lsof -i
```

Either way, you get a list of the processes on your Mac that currently have internet connections, along with the domain names or IP addresses to which they're connected, the ports they're using, and other useful tidbits.

Determine Whether Another Computer Is Running

Did your server crash? Is another Mac on your network turned on and awake? The easy way to find out if another computer is on, awake, and connected to the network is to use the `ping` command.

Enter this (substituting the remote computer's domain name or IP address):

```
ping address
```

If you see a sequence of lines that look something like this, the computer is responding:

```
64 bytes from 173.255.252.203: icmp_seq=0 ttl=49 time=21.27 ms
```

Press Control-C to stop pinging. If more than 30 seconds go by without any such line appearing, either the computer is offline, it is configured not to respond to pings, or there's a network problem.

Get Information About an Internet Server

What's the IP address of that web server you're connecting to? An easy way to find out is to use the host command:

```
host domain-name
```

This command returns the IP address(es) for that domain name. It also indicates if the domain name is an alias to another computer, and it lists any mail exchange servers associated with that domain. For example, enter `host www.takecontrolbooks.com` to learn the IP addresses of the Take Control web server, the fact that `www.takecontrolbooks.com` is actually an alias (pointer) to `takecontrolbooks.com`, and the domain name and IP address of the `takecontrolbooks.com` mail exchange server.

Alternatively, you can use a command called `nslookup` ("name server lookup") command, which takes either a domain name or an IP address as an argument:

```
nslookup takecontrolbooks.com
```

```
nslookup 173.255.252.203
```

In addition to showing you a host's domain name or IP address, `nslookup` gives you the IP address of the DNS server it consults, which can be handy to know if you're trying to diagnose a DNS problem.

Note: A newer command, called `dig` (domain information groper—yes, really), works much the same way and can also supply the IP address of a domain name, but requires special flags to do reverse lookups of domain names from IP addresses and presents it output in a much less readable form than `nslookup` or `host`.

Get Information About a Domain Name

If you need to find out what person or organization owns a domain name, enter the following:

```
whois domain-name
```

For example, if you enter `whois takecontrolbooks.com`, you'll learn which registrar handles the domain, the addresses of its name servers, and (perhaps) contact information for that domain's owner. (Many domain registrations omit owner contact information to preserve privacy.)

Flush Your DNS Cache

When you connect to any internet service (a web server, an email server, the iTunes Store, or whatnot), your Mac asks a DNS server to convert the server name (like takecontrolbooks.com) into an IP address (like 173.255.252.203). Your Mac then stores that IP address for a while in a DNS cache, so that the next time you connect to the same server, it can skip the DNS lookup step and get you there a bit faster.

However, sometimes server addresses change, and sometimes your DNS cache can get corrupted. In either case, you might find yourself connecting to the wrong site (or not connecting at all). The simplest solution is to flush the DNS cache, forcing macOS to look up IP addresses from scratch the next time you try to connect to each server by name.

The way you do this varies depending on your version of macOS:

- 10.10.4 Yosemite and later: `sudo killall -HUP mDNSResponder`
- 10.10.0–10.10.3 Yosemite: `sudo discoveryutil mdnsflushcache`
- 10.7 Lion–10.9 Mavericks: `sudo killall -HUP mDNSResponder`
- 10.6 Snow Leopard: `sudo dscacheutil -flushcache`

Note that since these commands use `sudo`, you'll have to supply an administrator password.

Do More with SSH

I covered the basics of using SSH to log in to another Mac and run commands there in Start an SSH Session. Here, I wanted to tell you about two slightly more advanced things you can do with SSH.

Verify an RSA Fingerprint for SSH

In step 2 of Start an SSH Session, I said you must confirm that the fingerprint you're seeing for a remote computer matches the one it's supposed to have. But how can you know what the computer's fingerprint is supposed to be?

You can ask the administrator of the remote computer, if there is one. But if the computer is one of your own (or you at least have physical access to it), you can determine its fingerprint with the following commands, which you enter in Terminal on that Mac:

- 10.11 El Capitan and later:
  `ssh-keygen -l -f /private/etc/ssh/ssh_host_rsa_key.pub`

- 10.8 Mountain Lion through 10.10 Yosemite:
  `ssh-keygen -l -f /private/etc/ssh_host_rsa_key.pub`

You can see the only difference in the commands is the path to the file containing the key. If the Mac is running a different operating system or a different version of macOS, the key might be located in another place, so you'll have to find it—either using a command such as `find` or `locate`, or by searching the web to find the key location for that particular operating system.

Set Up a Passwordless SSH Login

Ordinarily, you must enter your password for the remote computer when you log in with SSH. However, as an alternative, you can use a public-private key pair. Once you've set this up, you enter your login command as usual (e.g., `ssh jk@remote-Mac.com`), but that's it—the remote computer logs you in immediately without asking for your password.

I should point out that depending on your circumstances, this may be either more or less secure than using a password. On the plus side: it avoids the possibility of a keylogger slurping up your password when you log in—and it can give you a way to access remote computers that disallow password-based logins for security reasons. On the minus side, if anyone else can access your Mac's user account, there's no barrier to them logging in to remote servers this way. So: use with caution.

To set up passwordless login, follow these steps:

1. Create a `.ssh` directory (if one doesn't already exist) in your home directory, and move into it—that's the standard location for storing SSH keys:
   ```
   mkdir ~/.ssh; cd ~/.ssh
   ```
   Note that because this directory starts with a dot (.) it will normally be invisible in the Finder.

2. Apply the correct permissions to that directory:
   ```
   chmod go-rwx .ssh
   ```

3. Create a public-private key pair:

   ```
   ssh-keygen -b 1024 -t rsa -f id_rsa -P ""
   ```

   This process puts two new files in `~/.ssh`: `id_rsa` (that's your private key, which you should never share), and `id_rsa.pub` (that's your public key, which you'll copy to the other computer.

4. Create an empty file called authorized_keys that will hold keys from other servers:

   ```
   touch authorized_keys
   ```

5. Put the contents of your public key on the clipboard, using the Redirect Input (<) technique:

   ```
   pbcopy < ~/.ssh/id_rsa.pub
   ```

6. Log in to the remote computer using SSH as you normally do.

7. On the remote computer, switch to the `~/.ssh` directory:

   ```
   cd ~/.ssh
   ```

8. Open the remote computer's authorized_keys file:

   ```
   nano ~/.ssh/authorized_keys
   ```

9. Use the arrow keys to go down to an empty line at the end of the file and press ⌘-V to paste the public key you copied in step 5.

10. Press Control-X to exit, press `Y` to save, and press Return to accept the existing filename.

11. Enter `exit` to close the remote SSH connection.

Now, try logging in again, as you did in step 6—but this time (and from now on), you won't be prompted for your password!

You can repeat this process for as many remote computers as you like; just skip the first four steps (since you can reuse the same key pair) and start with step 5.

Tip: 1Password 8 has an SSH key management feature that can compress all of the above into just a few steps. It lets you unlock SSH sessions with Touch ID (on supported Macs) and syncs your keys across devices, too. For details, read *Take Control of 1Password*.

Work with Remote Macs

These two recipes help you work with Macs you’re accessing remotely.

Restart a Remote Mac

If you need to reboot the Mac you’re sitting in front of, you can simply choose Apple > Restart.

But what if you’re logged in to another Mac via SSH? To restart it, just enter this:

```
reboot
```

Needless to say, you should use this with caution—anyone else who happens to be using the computer at the time might be unhappy!

Note: You can use this command to reboot your own Mac, too, but it’s usually safer to choose Apple > Restart.

Restart a FileVault-Protected Mac Without a Password Prompt

If you need to remotely reboot a Mac that’s protected with FileVault, it’s possible to do so without the Mac getting stuck on the password screen when it turns back on.

First, make sure the remote Mac supports the `authrestart` command. You can check by connecting to the remote Mac via SSH (or, if you have physical access to the Mac, opening Terminal on it) and entering:

```
fdesetup supportsauthrestart
```

Note: The *fde* in `fdesetup` stands for Full Disk Encryption.

If that command returns `true`, you’re good to go.

The command to restart the system immediately without a password prompt afterward is:

```
sudo fdesetup authrestart
```

Enter that, supply your administrator password, and the remote Mac should reboot without any further fuss.

Troubleshoot and Repair Problems

These next few recipes can help you solve problems that keep your Mac from working correctly.

Delete Stubborn Items from the Trash

Occasionally, you may find that no matter what you do, you can’t empty your Trash. Maybe you get an inscrutable error message, or maybe it just doesn’t work. If this happens and you’re running 10.10 Yosemite or earlier, the first thing to try is choosing Finder > Secure Empty Trash (which was removed in 10.11 El Capitan, alas). If that doesn’t work, however, try the following (taking all the necessary precautions associated with `sudo`, of course):

```
sudo rm -ri ~/.Trash/*
```

The `rm` command prompts you for confirmation to remove each item.

> ***Warning!*** Do be certain to type these commands exactly right; using `sudo rm` can do some serious damage if you’re not careful!

If that doesn’t work, try each of these until the Trash is empty:

```
sudo rm -ri /.Trashes/*
```

```
sudo rm -ri /Volumes/*/.Trashes/*
```

Tip: If even `sudo rm` won’t delete a file or folder—whether or not it’s in the Trash—try entering `sudo chflags -R nouchg,noschg path/to/item`; then retry `sudo rm`. `Chflags` is “change flags,” a way of assigning or removing certain kinds of metadata. `Schg` is the system “immutable” flag, and `uchg` is the user “immutable” flag. These are much like the “locked” checkbox in the Finder’s Get Info window. So `noschg,nouchg` means “take away those flags if they exist.”

Figure Out Why You Can’t Unmount a Volume

Have you ever tried to eject an external hard drive, disk image, or network volume, only to see an error message saying the volume is in use? If so, the maddening part can be figuring out *which* process is using it so you can quit that process. So enter the following, substituting for *VolumeName* the name of the volume you can’t unmount:

```
lsof | grep /Volumes/VolumeName
```

This command shows you any processes you own that are currently using this volume; armed with this information, you can quit the program (using the `kill` command if necessary, as described in Stop a Program). One frequent offender: the `zsh` or `bash` shell itself! If that’s the case, you’ll see something like this:

```
zsh  14384  jk  cwd  DIR  45,8  330  2 /Volumes/Data
```

If you’ve navigated to a directory on this volume in your shell, macOS considers it “in use.” The solution in this case is to exit the shell, or simply `cd` to another directory.

If this command doesn’t tell you what you need to know, repeat it, preceded by `sudo`.

Find Out What's Keeping Your Mac Awake

If your Mac refuses to sleep, it's likely because some process is completing a task that prevents sleep from occurring immediately. But which process would that be? To find out, try this command:

```
pmset -g assertions | grep -E "(PreventUserIdleSystemSleep|
PreventUserIdleDisplaySleep)"
```

The output will look something like this:

```
PreventUserIdleDisplaySleep    0
PreventUserIdleSystemSleep     1
pid 550(sharingd): [0x0001120f000190bb] 00:01:59
PreventUserIdleSystemSleep named: "Handoff"
```

The first line (`PreventUserIdleDisplaySleep`) tells you if anything is preventing *display* sleep (`0` for no, `1` for yes), and the second line (`PreventUserIdleSystemSleep`) tells you if anything is preventing *system* sleep.

If the answer to either of these is `1` (yes), the line below lists the PID and name of that process—in this example, `sharingd` (the system process responsible for Handoff), although you may also see Time Machine or any of numerous other processes.

Tip: What if you have the opposite problem and you *want* to keep your Mac awake for some reason? No problem: just enter `caffeinate` and your Mac will stay awake until you press Control-C or end your shell session. To keep your Mac awake for a specific amount of time, enter `caffeinate -u -t 7200` (`7200` is two hours expressed in seconds: you can use whatever time you want).

Reset the Launch Services Database

The Launch Services database in macOS keeps track of which programs are used to open which files, among other things. If you find that the wrong app opens when you double-click files, or that icons don't match up with the correct items, you may have to reset your Launch Services database. Do it like this (enter the command as a single, long line—no space between `LaunchServices.` and `framework` and omit hyphens that you may see in the path):

```
/System/Library/Frameworks/CoreServices.framework/Frameworks/
LaunchServices.framework/Support/lsregister -kill -r -domain local
-domain system -domain user
```

Because this resets a lot of default mappings, your Mac may think you're launching apps for the first time and ask if it's OK. Agree to the alerts and you should be in good shape.

Use Disk Utility from the Command Line

Apple's venerable Disk Utility app, included with macOS, can diagnose and fix many types of disk problems. It can also mount and unmount volumes; erase, format, and partition disks; create and convert disk images; and more. Should you ever have need of these capabilities on the command line (for example, when accessing a remote Mac via SSH, or if Disk Utility isn't working), you can access them with `diskutil`, the command-line version of Disk Utility.

The list of tasks `diskutil` can perform is quite long—just take a gander at `man diskutil` and you'll see what I mean. In fact, diskutil can even do a bunch of things that the GUI version of the app can't, such as creating Fusion volumes (something I don't cover in this book). Because you'll most likely use Disk Utility except on rare occasions, I'm not going to go into much detail about `diskutil`. However, I do want to acquaint you with a few important basics.

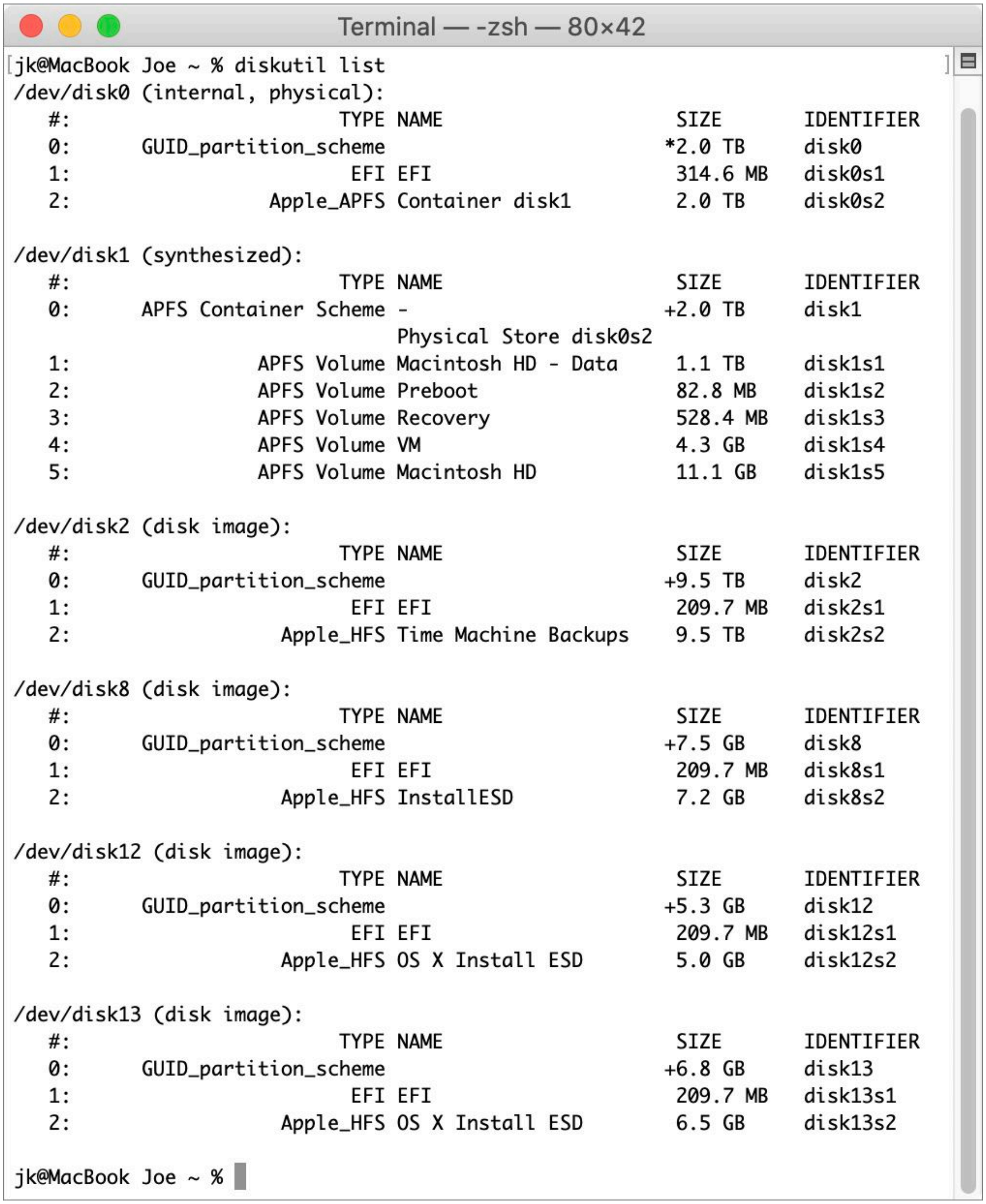
List Volumes

Nearly every task in `diskutil` requires you to identify the disk or volume you want to work on. In some cases, but not all, the visible volume name (like "Macintosh HD") works. Even when it does, it's more precise to use a code called a *disk identifier*. So the first thing you'll likely want to do is list all your *devices* (disks and volumes) to get their disk identifiers.

Start with:

```
diskutil list
```

That command spits out a list of available devices—many of which may not be visible in the Finder. For example, on a MacBook Pro running Catalina—without any manual partitions—I see six devices, each with multiple volumes, called *slices* (**Figure 12**).



```
jk@MacBook Joe ~ % diskutil list
/dev/disk0 (internal, physical):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        *2.0 TB     disk0
   1:                        EFI EFI                     314.6 MB   disk0s1
   2:                 Apple_APFS Container disk1         2.0 TB     disk0s2

/dev/disk1 (synthesized):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      APFS Container Scheme -                      +2.0 TB     disk1
                                 Physical Store disk0s2
   1:                APFS Volume Macintosh HD - Data     1.1 TB     disk1s1
   2:                APFS Volume Preboot                 82.8 MB    disk1s2
   3:                APFS Volume Recovery                528.4 MB   disk1s3
   4:                APFS Volume VM                      4.3 GB     disk1s4
   5:                APFS Volume Macintosh HD            11.1 GB    disk1s5

/dev/disk2 (disk image):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        +9.5 TB     disk2
   1:                        EFI EFI                     209.7 MB   disk2s1
   2:                  Apple_HFS Time Machine Backups    9.5 TB     disk2s2

/dev/disk8 (disk image):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        +7.5 GB     disk8
   1:                        EFI EFI                     209.7 MB   disk8s1
   2:                  Apple_HFS InstallESD              7.2 GB     disk8s2

/dev/disk12 (disk image):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        +5.3 GB     disk12
   1:                        EFI EFI                     209.7 MB   disk12s1
   2:                  Apple_HFS OS X Install ESD        5.0 GB     disk12s2

/dev/disk13 (disk image):
   #:                       TYPE NAME                    SIZE       IDENTIFIER
   0:      GUID_partition_scheme                        +6.8 GB     disk13
   1:                        EFI EFI                     209.7 MB   disk13s1
   2:                  Apple_HFS OS X Install ESD        6.5 GB     disk13s2

jk@MacBook Joe ~ % 
```

Figure 12: Example output from `diskutil list`.

Look through this list and find the volume you're interested in. The NAME column will often help, although you may have to make some inferences from the information in the TYPE and SIZE columns too. In this example, the identifier for my read-only system volume, Macintosh HD, is `disk1s5` (that's disk 1, slice 5). The Macintosh HD - Data volume, where most of my data is stored, is `disk1s1`. And the APFS container that holds both of those (and a few others) is just plain `disk1`.

Once you have the identifier for the disk you want to work with, you can plug it into various commands. I'll list a few here, pretending that I want to use them on `disk1s1`. (But remember, don't use these as is—substitute the identifier for whatever disk you want to use.)

Unmount a Mounted Volume

To umount (or eject) a mounted volume with the identifier `disk1s1`, enter:

```
diskutil unmount disk1s1
```

> **Note:** Unless you're running Terminal in recovery mode, you can't umount your startup volume.

To unmount all the volumes (slices) on `disk1` at once, enter:

```
diskutil unmountDisk disk1
```

Mount an Unmounted Volume

If `disk1s1` is available but unmounted, I can mount it like so:

```
diskutil mount disk1s1
```

To mount all the volumes (slices) on `disk1` at once, enter:

```
diskutil mountDisk disk1
```

Verify a Volume

To check the volume `disk1s1` for errors, but without trying to repair any that are found, enter:

```
diskutil verifyVolume disk1s1
```

Just as in the previous examples, `diskutil verifyDisk disk1` would do the same thing for the whole disk.

Repair a Volume

To check the volume disk1s1 for errors *and* attempt to repair any that are found, enter:

```
diskutil repairVolume disk1s1
```

Once again, diskutil repairDisk disk1 would do the verify and repair operation on the whole disk.

Note: When you use diskutil verifyVolume or diskutil repairVolume, behind the scenes, a program called fsck ("file system consistency check") runs. You can also run that program by itself in some situations—see the next recipe.

For many more options, consult man diskutil.

Fix Disk Problems in Single-User Mode

In the previous recipe, I mentioned the fsck tool, which Disk Utility relies on for certain tasks. You can also run it separately, but only when your Mac is booted in single-user mode. Only older, Intel-based Macs can do this—if your Mac has a T2 chip (see this list) or an M-series Apple silicon processor, single-user mode is unavailable without extensive and dangerous fiddling. (If you have a newer Mac, you can do something comparable in recovery mode; see the next two recipes.)

First, power on (or restart) your Mac while holding down ⌘-S to enter single-user mode. You'll see a text display much like the inside of a Terminal window.

Enter the following command:

```
/sbin/fsck -y
```

The fsck tool checks most of your disk for errors, and repairs those it can. To restart your Mac normally afterward, enter exit.

Reboot in Recovery Mode

Earlier, in Restart a Remote Mac, I mentioned the reboot command, which restarts whichever Mac you run it on (whether local or remote).

If you want to reboot in recovery mode (for troubleshooting purposes, or to reinstall macOS, for instance), the normal way to do this is to hold down ⌘-R while restarting or, on M-series Macs, press and hold the power button (see About Recovery Mode for details). But it's also possible to tell your Mac that its next boot should go directly into recovery mode, no special keys required.

Here's how:

```
sudo nvram recovery-boot-mode=RecoveryModeDisk
```

Note: The nvram command refers to "non-volatile RAM," a type of RAM whose contents are preserved when your Mac is powered off.

After you enter this command, you can restart in the usual way or enter the reboot command. Note, however, that until you change it, your Mac will *always* be set to restart in recovery mode! Thus, while you're in that mode, it's essential that you choose Utilities > Terminal and enter this command to reset your Mac to normal booting:

```
sudo nvram -d recovery-boot-mode
```

There are several variations on the reboot-in-recovery-mode trick, by the way. To reboot into *internet* recovery, use:

```
sudo nvram internet-recovery-mode=RecoveryModeNetwork
```

Or, to reboot into Apple Hardware Diagnostics, use:

```
sudo nvram recovery-boot-mode=DiagsModeDisk
```

Use Terminal in Recovery Mode

Once you've rebooted into recovery mode (in the normal way, as described in About Recovery Mode, or using the command in the previous recipe), you'll see a window showing several options. There's also a Utilities menu, and one of the command on it is Terminal, so (as you'd expect), you can choose Utilities > Terminal to run Terminal.

However, if you plan on doing anything in Terminal that requires access to any of your disks, you must mount them *before* you open

Terminal. To do so, select Disk Utility and click Continue. Select the volume you want to mount in the sidebar. Then click Mount and supply your password when prompted. Repeat this for any additional volumes you want to mount.

Now you can choose Utilities > Terminal!

However, be aware of the following:

- Terminal runs the `bash` shell in recovery mode—`zsh` is not available.
- Your `.zshrc` or `.bash_profile` won't load, because you're not logged in under your regular user account.
- Because your PATH won't include the directories you expect, it's always best to run programs using their *full* pathname. But just as zsh is missing in recovery mode, so are many other Unix programs. To reference a program on your startup disk, supply its full pathname starting with `/Volumes`—for example, `/Volumes/Macintosh\ HD/usr/bin/rsync`.

When you're finished in Terminal, you can `exit` as usual to use other macOS utilities, or enter `reboot` to restart.

Turn Off Auto Boot

If you have a Mac laptop introduced in 2016, it most likely includes a hidden and little-discussed feature called Auto Boot. Simply put, this feature turns on the computer, even if it's fully shut down, when you press *any* key. I suppose that makes turning on the computer slightly easier than having to press the power button in particular, but there's a consequence: if you want to, say, clean the keyboard, replace a key cap, or do any other sort of maintenance or repair that might involve touching a key, you'll be unable to do so without turning on the Mac!

There's a simple solution, however:

```
sudo nvram AutoBoot=%00
```

Once you've entered that, you can shut down your Mac and not worry that it will turn itself on—you'll have to press the actual power button to do so.

To reenable Auto Boot, use this:

```
sudo nvram AutoBoot=%03
```

Enable or Disable Screen Sharing Remotely

I frequently use screen sharing to view and control the screens of other Macs I own. But sometimes—for reasons I can't explain even though this has been going on through many updates of macOS—one of my Macs decides to stop responding when I try to share its screen. If I go to the Mac in person and open System Settings > General > Sharing, Screen Sharing appears to be on, but that Mac doesn't show up in the Finder of other Macs on my local network as being available for screen sharing. I hear other people have had similar experiences.

Fortunately, as long as another Mac is available for remote login via SSH (see Log In to Another Computer), you can fix this problem quickly and easily. (For that matter, if you have SSH access, you can use this to turn screen sharing on even if it was turned off before.) Do this:

1. Connect to the remote Mac via SSH.
2. To disable screen sharing (which you might have to do even if it's on but, as in my example above, not responding), enter:
   ```
   sudo launchctl bootout system /System/Library/LaunchDaemons/
   com.apple.screensharing.plist
   ```

   Enter your administrator password when prompted.
3. Then, to (re)enable screen sharing, enter:
   ```
   sudo launchctl bootstrap system /System/Library/LaunchDaemons/
   com.apple.screensharing.plist
   ```

As soon as you've done that, you should be able to share the other Mac's screen.

Get Help in Style

The next two recipes let you read `man` pages in a friendlier environment than Terminal, without installing any extra software. After that is a technique programmers can use to get instant advice.

Read man Pages in Preview

Earlier versions of this book had a terrific recipe here to view `man` pages in Preview, relying on the twin facts that man can save manual pages as beautifully formatted PostScript files, and Preview could read those files. For reasons I can only guess at, Apple removed PostScript compatibility from Preview in Ventura, so that old recipe doesn't work anymore. However, there's another way to get the same result, by using a command called `mandoc` to convert man pages to PDF format (which Preview *can* still read). And, since that command also works in earlier versions of macOS, you can use this revised recipe in Catalina or later.

Because it's a multi-step process, you need a shell function (like a shell script, but contained directly in `.zshrc` or `.bash_profile` file) to help you do this. So, following the instructions in Customize Your Defaults, put the following lines in `.zshrc` or `.bash_profile`:

```
preman()
{
mandoc -T pdf "$(/usr/bin/man -w $@)" | open -f -a Preview.app
}
```

Having done that—and after reloading your new configuration by entering `source .zshrc` (for `zsh`) or `source .bash_profile` (for `bash`), or by opening a new shell—enter the following, substituting the name of whatever command you want to read about in Preview:

```
preman command
```

Et voilà! After a few seconds, a spiffy manual page opens in Preview.

Read man Pages in BBEdit

Perhaps you're a plain text, monospaced font kind of person. If you keep BBEdit open anyway, you can use it to open man pages instead of the built-in man program.

To make this happen, install the command-line tools available for either editor, add the following line to `.zshrc` or `.bash_profile` (see Customize Your Defaults), and then start a new shell session:

```
export MANPAGER="col -b | bbedit --clean --view-top"
```

Thereafter, entering `man` (followed by the command of your choice, such as `man ls`) displays the manual page in your text editor.

Get Quick Answers to Programming Questions

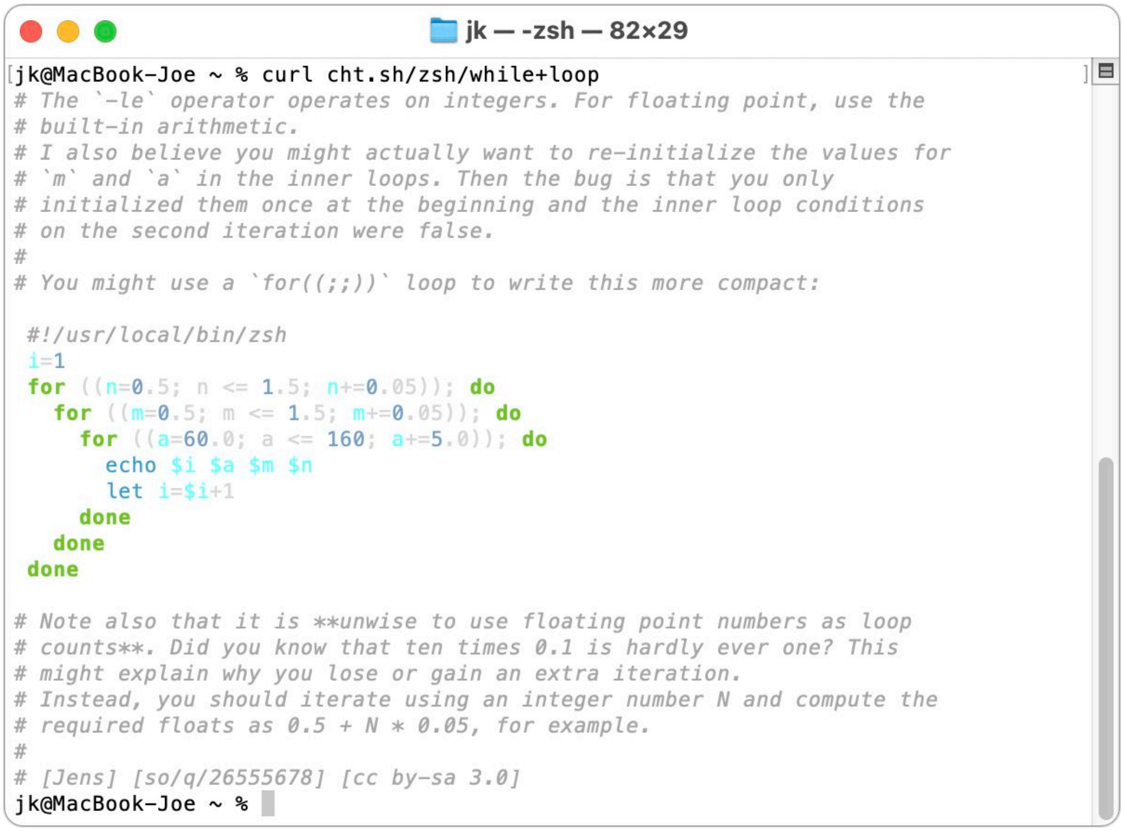
Whether you write apps in Swift or Objective-C, program for the web in PHP or JavaScript, or simply write the occasional `zsh` or `bash` script, you're bound to encounter programming puzzles, problems, and questions from time to time. You might head to your favorite search engine searching for hints, or you might head directly to one of the most popular online communities for sharing programming advice: Stack Overflow.

A clever developer named Igor Chubin has created a tool called `cheat.sh` (as in cheat sheet—also available in an abbreviated form, `cht.sh`) that takes a question you enter on the command line, looks it up on Stack Overflow, and returns the first answer it finds—again, right on the command line! This can be used for virtually any programming or scripting language.

The format to use is: `curl cht.sh/language/query` (where spaces in the query are replaced with `+` signs). For example, let's say I want to know how to write a *while* loop in `zsh`. I can try:

```
curl cht.sh/zsh/while+loop
```

And here's what I get back (**Figure 13**):



```
jk@MacBook-Joe ~ % curl cht.sh/zsh/while+loop
# The `-le` operator operates on integers. For floating point, use the
# built-in arithmetic.
# I also believe you might actually want to re-initialize the values for
# `m` and `a` in the inner loops. Then the bug is that you only
# initialized them once at the beginning and the inner loop conditions
# on the second iteration were false.
#
# You might use a `for((;;))` loop to write this more compact:

 #!/usr/local/bin/zsh
 i=1
 for ((n=0.5; n <= 1.5; n+=0.05)); do
   for ((m=0.5; m <= 1.5; m+=0.05)); do
     for ((a=60.0; a <= 160; a+=5.0)); do
       echo $i $a $m $n
       let i=$i+1
     done
   done
 done

# Note also that it is **unwise to use floating point numbers as loop
# counts**. Did you know that ten times 0.1 is hardly ever one? This
# might explain why you lose or gain an extra iteration.
# Instead, you should iterate using an integer number N and compute the
# required floats as 0.5 + N * 0.05, for example.
#
# [Jens] [so/q/26555678] [cc by-sa 3.0]
jk@MacBook-Joe ~ %
```

Figure 13: An example answer to a query to `cht.sh`.

The graphic above may be a bit difficult to read, but the gist of it is that it's the answer to a specific question some programmer had about doing something with a *while* loop in `zsh`. The specific question you have might not be the same, but if all you were looking for was the basic syntax, you could see it clearly enough in the example.

But remember, that's just the first answer to a Stack Overflow search for those terms. If you want to see the next answer, append `/2`, like so:

```
curl cht.sh/zsh/while+loop/2
```

And, of course, you can keep going to find additional answers.

You can also ask cht.sh about specific commands (for example, `curl cht.sh/pwd` or `curl cht.sh/scp`) or even learn the basics of an entire programming language (for example, `curl cht.sh/swift/:learn`). For more syntax examples, see cheat.sh.

Do Other Random Tricks

Finally, we have a bunch of interesting recipes that didn't fit neatly in any of the other categories. Enjoy!

Unhide Your User Library Folder

I've mentioned the user Library folder (that's `/Users/you/Library` or `~/Library` for short) numerous times in this book. It's never a problem to see that on the command line, but Apple has been hiding it in the Finder since the days of 10.7 Lion. They do this, I'm sure, to reduce the risk of users accidentally causing damage. But it's a pity, because in knowledgeable hands, the `~/Library` folder is extremely useful, and I often find myself visiting it even when not in Terminal.

There are *at least* 19 different ways to view this folder (see 18 ways to view the ~/Library folder in Lion and Mountain Lion at Macworld, which was later updated with a 19th technique). Here's the one I use—I did it once, many years ago, and my `~/Library` folder has remained unhidden in the Finder through many upgrades of macOS:

```
chflags nohidden ~/Library
```

That's it! Once you do this, your `~/Library` appears within your home folder as well as on the Go menu in the Finder.

Take a Screenshot

You can take a screenshot by pressing ⌘-Shift-3; the image is named Picture *X* by default and stored on your desktop. But what if you want to take a screenshot and give it a different name, or store it somewhere else? Or—this is pretty cool—take a screenshot of a remote Mac you're logged in to via SSH?

You can do it with this command (substituting the name and location of the saved screenshot to taste):

```
screencapture ~/myscreen.png
```

Note: Recall that, starting in Catalina, the `screencapture` program requires special permission in System Settings > Privacy & Security > Screen Recording or System Preferences > Security & Privacy > Privacy > Screen Recording.

Compact a Disk Image

Of the numerous disk image formats Disk Utility can create, two of them—sparse disk images and sparse bundle disk images—have the interesting characteristic that they can expand as needed (up to a preset limit) to accommodate more files. (See my TidBITS article Discovering Sparse Bundle Disk Images.) The only problem is, they don't automatically shrink again when you delete files!

The `hdiutil` program can take care of this for you easily. To compact a sparse or sparse bundle image so that it takes up only the space it needs, enter the following, substituting the image's name and location as appropriate:

```
hdiutil compact image.dmg
```

Note: The `hdiutil` tool can do lots more with disk images—create them, convert them, encrypt them, and so on. See `man hdiutil`.

Use Text-to-Speech from the Command Line

This is mostly just for fun. To have your Mac speak text using the text-to-speech voice currently selected in System Settings > Accessibility > Spoken Content or System Preferences > Dictation & Speech (Speech in older versions of macOS) > Text to Speech, enter the following:

```
say "Hello there"
```

Note that this even works remotely, assuming you're logged in to a Mac on the other end. Use your power responsibly!

Note: Under Sonoma and later, this command may fail if you have enabled Personal Voice (in System Settings > Accessibility > Personal Voice).

As a more practical example, you can make a quick-and-dirty countdown timer using a command like this, substituting for `60` the number of seconds to wait before the Mac speaks the text you enter:

```
sleep 60; say "One minute has elapsed"
```

Enable or Disable Your Mac’s Startup Chime

Starting in 2016, Apple disabled the startup chime by default in most new Mac models…then, in Big Sur, they brought it back! (Now, you can enable or disable the startup chime by enabling or disabling “Play sound on startup” in System Settings > Sound or System Preferences > Sound > Sound Effects.) If you have a recent-vintage Mac that *isn’t* running Big Sur or later, and you miss the startup chime, use this to enable it:

```
sudo nvram StartupMute=%00
```

To mute the startup chime on a modern Mac (even under Big Sur or later, as an alternative to the System Settings/System Preferences control), use this:

```
sudo nvram StartupMute=%01
```

On an older Mac that normally has a startup chime, you can use this command to mute it:

```
sudo nvram SystemAudioVolume=%80
```

To reenable the chime on older Macs, you use a somewhat different command:

```
sudo nvram -d SystemAudioVolume
```

Send an SMS from the Command Line

You can send an SMS text message from a phone, and with the combination of Yosemite or later and an iPhone, you can send an SMS using Messages on your Mac. But even without an iPhone—and with any version of macOS—you can send an SMS from the command line! This might be useful in a shell script—for example, if you wanted a notification that a script completed successfully (or failed).

This command uses a service from Ian Webster called TextBelt (consult the TextBelt site for more information):

```
curl -X POST http://textbelt.com/text -d number=mobile-number -d
message="message-text" -d key=textbelt
```

Fill in *mobile-number* with the 10-digit phone number (consult the TextBelt site for details on using the service outside the United States) and replace *message-text* with your message text.

Although the service used to be entirely free, now you can send only one free message per day. To send more, you must pay a modest fee, which is then associated with a custom API key you generate on the site. That key then replaces *textbelt* in the command above.

Find Item Frequency in a CSV File

OK, this one is a bit obscure, but hear me out. I frequently work with CSV (comma-separated values) files, such as exported spreadsheets and databases. Here's an example of what the content might look like:

```
Ann,2020,purple,42
Bob,2019,green,13
Carla,2013,blue,42
David,2011,purple,13
Edie,2020,green,42
```

(And no, that data isn't a secret code or anything—it's just random.)

What if I simply wanted to find the frequency of the color purple (from the third column) or the amount (from the fourth column)? Assuming the above data is in a file called `file.csv`, I'd do it like this, where # is replaced with the column I want to sort on (credit: Bruce Potter):

```
cut -d ',' -f # file.csv | sort -n | uniq -c | sort -nr
```

For example, to sort by color (column 3), I'd use:

```
cut -d ',' -f 3 file.csv | sort -n | uniq -c | sort -nr
```

That command, applied to that file, would yield the following output:

```
2 purple
2 green
1 blue
```

That is: two instances each of column 3 contain the color purple, and then there's one of blue.

Or, to sort by amount (column 4), I'd use:

```
cut -d ',' -f 4 file.csv | sort -n | uniq -c | sort -nr
```

That command, applied to that file, would yield the following output:

```
3 42
2 13
```

In other words: three instances of 42, and two instances of 13.

Granted, that exact maneuver is one you may rarely if ever have need of, but I wanted to show you how this recipe works so you can adapt it to your needs.

As you can see, it's four commands feeding sequentially into each other with pipes (`|`)—the output of `cut` goes to `sort`, and the output of `sort` goes to `uniq`, and the out of `uniq` goes to `sort` again. Here's how this breaks down:

- `cut -d ',' -f 3 file.csv`
  The `cut` command cuts (or, really, copies) an element from a line in a file, identifying it by the *delimiter* (`-d`) specified within single quotation marks (a comma). The `field` (`-f`) argument tells it that the third such item is what we want, and of course `file.csv` is the file we're looking in.

So, this part of the command, by itself, would yield the third item from each row, like so:

```
purple
green
blue
purple
green
```

- `sort -n`
  The `sort` command—you'll never guess—*sorts* the input! The `-n` flag means numeric sort (as opposed to, say, alphabetical). So, if the whole command to this point is run, you'll get:

  ```
  blue
  green
  green
  purple
  purple
  ```

- `uniq -c`
  The `uniq` command, which of course means "unique," filters out repeated lines. The `-c` flag means to precede each line with the number of times it appeared. So the output up to this point would be:

  ```
  1 blue
  2 green
  2 purple
  ```

- `sort -nr`
  Now we have a second instance of `sort`, but this time adding the `-r` (reverse) flag to `-n`, so the new list will be in descending order:

  ```
  2 purple
  2 green
  1 blue
  ```

When you see any multi-step command like this, I encourage you to break it down step by step, trying each element one at a time and checking out the relevant man pages as you go, to learn what each step does. Then try mixing and matching to make new commands!

Tip: If you use this sort of command a lot, rather than memorizing it, you can Use Global (Inline) Aliases for most of it.

Prevent a Laptop from Waking up When Jostled During Travel

Mac laptops are designed to go to sleep automatically when you close the lid and wake up automatically when you open the lid. But if your computer happens to be bumped just the right way while it's in its case, the lid can open just enough to wake up the computer, potentially causing it to overheat, or causing your battery to run down, while it should be asleep.

To prevent the computer from waking up automatically when the lid opens, enter this:

```
sudo pmset -a lidwake 0
```

Thereafter, wake your Mac by pressing a key after you open the lid.

(To reverse this setting, repeat the command, replacing the 0 with a 1.)

List More Directory Information

You should be thoroughly familiar with the ls ("list") command, introduced in See What's Here. Among the flags that modify its behavior, I've described elsewhere in this book -l ("long format") and -h ("human-readable").

But if you want ls to truly show you everything, you need to add a few more flags.

To make the command easier to use, add an alias to `.zshrc` or `.bash_profile` (see Create Aliases) like this:

```
alias ls="ls -lah@e"
```

The flag `-a` lists all files, including hidden ones (those that begin with a period). The `-@` flag lists extended attributes (indicated by an `@` at the end of a permissions string), and the `-e` flag lists all access control lists, or ACLs (indicated by a `+` at the end of a permissions string). (And yes, I agree that the meanings of `-@` and `-e` seem backward at first glance!)

Tip: For frequent *very brief* command-line recipes, follow Mark Krenz's Command Line Magic Mastodon account.

About This Book

Thank you for purchasing this Take Control book. We hope you find it both useful and enjoyable to read. We welcome your comments.

Ebook Extras

You can access extras related to this ebook on the web. Once you're on the ebook's Take Control Extras page, you can:

- Download any available new version of the ebook for free, or buy a subsequent edition at a discount.
- Access the book in both PDF and EPUB formats. (Learn about reading on mobile devices on our Device Advice page.)
- Read the ebook's blog. You may find new tips or information, as well as a link to an author interview.
- Find out if we have any update plans for the ebook.

If you bought this ebook from the Take Control website, it has been automatically added to your account, where you can download it in other formats and access any future updates.

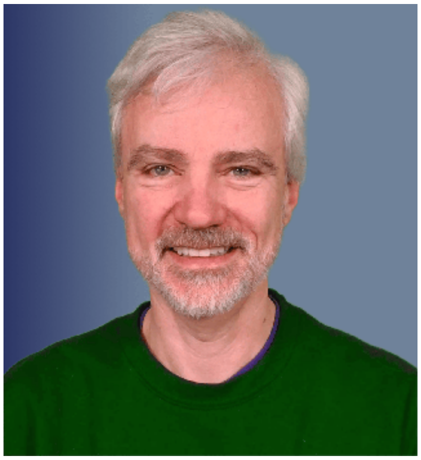
More Take Control Books

This is but one of many Take Control titles! We have books that cover a wide range of technology topics, with extra emphasis on Macs and other Apple products.

You can buy Take Control books from the Take Control online catalog as well as from venues such as Amazon and the Apple Books Store. But it's a better user experience and our authors earn more when you buy directly from us. Just saying…

Our ebooks are available in two formats, PDF and EPUB, which are viewable on any computer, smartphone, tablet, or e-reader. All are DRM-free.

About the Author and Publisher



Joe Kissell is the author of more than 60 books about technology. In 2017, he also became the publisher of Take Control Books, when alt concepts—the company he runs with his wife, Morgen Jahnke—acquired the Take Control series from TidBITS Publishing Inc.'s owners, Adam and Tonya Engst.

Joe previously wrote for TidBITS, Macworld, and Wirecutter (among other publications). Before he began writing full-time in 2003, Joe spent nearly eight years managing software development. He holds a bachelor's degree in Philosophy and a master's degree in Linguistics.

In his rare non-work hours, Joe likes to walk, cook, eat, and practice tai chi. He lives in Saskatoon, Saskatchewan, Canada, with Morgen and their sons. To contact Joe about this book, send him email and *please* include `Take Control of the Mac Command Line with Terminal` in the subject of your message. You can follow him on Mastodon (@joekissell) or Bluesky (@joekissell.com), or visit his personal website, JoeKissell.com.

Author's Acknowledgments

Thanks to Geoff Duncan for an outstanding editing job, and to the following people for suggesting command-line recipes: Marshall Clow, Keith Dawson, Geoff Duncan, Chuck Goolsbee, Jon Gotow, Kevin van Haaren, Andrew Laurence, Peter N Lewis, Chris Pepper, Larry Rosenstein, and Nigel Stanger.

Credits

- Publisher: Joe Kissell
- Editor: Geoff Duncan
- Cover design: Sam Schick of Neversink
- Logo design: Geoff Allen of FUN is OK

Also by Joe Kissell

Click any book title below or visit our web catalog to add more ebooks to your Take Control collection!

Take Control of 1Password: Use this powerful password manager to create, store, enter, and sync personal data on all your devices.

Take Control of Apple Mail: Learn the ins and outs of Apple's email app in macOS and iOS.

Take Control of Automating Your Mac: Work more efficiently on your Mac with time-saving shortcuts of all kinds.

Take Control of Backing Up Your Mac: Protect your Mac's valuable data from any sort of mishap.

Take Control of DEVONthink 3: Master this powerful information management tool.

Take Control of iCloud: Make the most of Apple's online service for storing, syncing, and sharing data.

Take Control of Sequoia: Discover what's new in macOS 15, and get all the information you need to upgrade safely.

Take Control of Your Digital Legacy: Make sure your important digital information is preserved for future generations.

Take Control of Your Online Privacy: Learn what's private online (not much)—and what to do about it.

Take Control of Your Paperless Office: With your Mac, scanner, and this ebook, you'll finally eliminate the chaos of overflowing paper.

Take Control of Your Passwords: Overcome password overload! Stay safe while eliminating the hassles and confusion of passwords.

Copyright and Fine Print

Take Control of the Mac Command Line with Terminal, Third Edition
ISBN: 978-1-990783-03-6

alt concepts, 419 8B-3110 8th St. East, Saskatoon, SK S7H 0W2 Canada

Why Take Control? We designed Take Control electronic books to help readers regain a measure of control in an oftentimes out-of-control universe. With Take Control, we also work to streamline the publication process so that information about quickly changing technical topics can be published while it's still relevant and accurate.

Our books are DRM-free: This ebook doesn't use digital rights management in any way because DRM makes life harder for everyone. So we ask a favor of our readers. If you want to share your copy of this ebook with a friend, please do so as you would a physical book, meaning that if your friend uses it regularly, they should buy a copy. Your support makes it possible for future Take Control ebooks to hit the internet long before you'd find the same information in a printed book. Plus, if you buy the ebook, you're entitled to any free updates that become available.

Remember the trees! You have our permission to make a single print copy of this ebook for personal use, if you must. Please reference this page if a print service refuses to print the ebook for copyright reasons.

Caveat lector: Although the author and alt concepts have made a reasonable effort to ensure the accuracy of the information herein, they assume no responsibility for errors or omissions. The information in this book is distributed "As Is," without warranty of any kind. Neither alt concepts nor the author shall be liable to any person or entity for any special, indirect, incidental, or consequential damages, including without limitation lost revenues or lost profits, that may result (or that are alleged to result) from the use of these materials. In other words, use this information at your own risk.

It's just a name: Many of the designations in this ebook used to distinguish products and services are claimed as trademarks or service marks. Any trademarks, service marks, product names, or named features that appear in this title are assumed to be the property of their respective owners. All product names and services are used in an editorial fashion only, with no intention of infringement. No such use, or the use of any trade name, is meant to convey endorsement or other affiliation with this title.

We aren't Apple: This title is an independent publication and has not been authorized, sponsored, or otherwise approved by Apple Inc. Because of the nature of this title, it uses terms that are registered trademarks or service marks of Apple Inc. If you're into that sort of thing, you can view a complete list of Apple Inc.'s registered trademarks and service marks.