Awesome Command Line

21 Commands to Get You Started With Your Computer's Magic Portal!

Christopher S. Jones

Awesome Command Line
by Christopher S. Jones

Published by Christopher S. Jones, Post Office Box 7201, Boulder, CO 80306

November 2025: First Edition

Typeset by the author using the Asciidoc® Language™ [https://asciidoc.org][1] and a sprinkling of custom Ruby code. All source code for the book is available as a *free and open source software project* [https://github.com/christopher-s-jones/awesome-command-line]. The body text font is EB Garamond by Georg Duffner and Octavio Pardo. The heading font is PT Sans Narrow by Alexandra Korolkova, Olga Umpeleva and Vladimir Yefimov, and released by ParaType. The fixed width code font is Adobe Source Code Pro by Paul D. Hunt. The cover page title font is Qwitcher Grypen by Robert Leuschke, with PT Sans Narrow. Illustrations and logo design are by the author using Sketch.

[1] AsciiDoc® and AsciiDoc Language™ are trademarks of the Eclipse Foundation, Inc. Asciidoctor® is a trademark owned by Dan Allen, Sarah White, and the project's individual contributors.

To Kimberly, with love and gratitude.
Thank you for leading the Dawn.

CONTENTS

Welcome to *Awesome Command Line*! I am so grateful that you have found this book, and that you're interested in exploring the seemingly secret world of computing from the command line!

This book is a reminder that you have superpowers at your fingertips! We are all familiar with the ways that we interact with our computers on a daily basis — *click*, *drag*, *drop*, *scroll*, *touch*, *tap*. The graphical nature of our screens and applications are absolutely amazing and indispensible. In this book, we explore the secret little gems available on all operating systems with another familiar way to interact — *type*! And the best part is that the magical commands we explore are largely derived from the English words that describe them, so acquainting ourselves with these commands is straight-forward with a bit of guidance. Once you become familiar with this small set of essential commands, adding new commands to your toolbox is even easier. With some simple dedication and practice, you will be able to enhance your computing workflow to be even more efficient and powerful. You'll manage your projects in ways that would otherwise be labor intensive when using your mouse. The intention of this book is to empower you in your creative journey by showing you how the *command line* is an awesome tool to get things done!

The UNIX® operating system developed by Bell Laboratories in the 1970s and 1980s was groundbreaking technology. When I first learned about *Linux*®—a Unix-inspired operating system for personal computers, I was fascinated by the freedom of it, and spent a substantial amount of time figuring out how to work with it via text commands. Apple® transitioned to a Unix-like operating system with *Mac OS X*, and Microsoft® introduced the Windows® Subsystem for Linux to bring a Unix-like environment to Windows®. I've been excited about these moves because we now have access to these incredible tools in very polished desktop environments. My intention with this book is to save you as much start-up time and energy as possible in learning the command line, and empower you to get the most out of the operating system of your choice, be it *macOS*®, *Linux*®, or *Microsoft*® *Windows*®. Have fun with it![1]

Who Should Read This Book

This book is written for everyone who loves computing! It's written as an introductory book to get you started with the infinite universe of text commands. The twenty one commands that I highlight are essential to building your foundation, and the command line concepts throughout each chapter solidify it further. This is intended as a launching point for everyone who likes to be organized and productive. Those in creative works—*writers*, *artists*, *designers*, *photographers*, and *editors*, and those beginning their journey with computers—*students*, early career *scientists*, *librarians*, aspiring *engineers*, *software developers*, *system administrators*, and *project managers*, can all learn the foundations and then pick up the commands that are specific to their course of study, craft, or industry. Those interested in joining the *decentralized web by running their own Bitcoin or Lightning node, Nostr relay, and other freedom technologies* will benefit from knowing these essentials. No matter your interests, this book is for those who love to learn!

Conventions Used in This Book

The following informational icons are used throughout the book:

This icon indicates an informational note.

This icon indicates a tip, suggestion, or a point of awesomeness.

This icon indicates a time to be very careful with a command or action. It could be irreversible.

Getting the Command Line Examples

The examples in this book are accessible for download from a link on the book release page [https://github.com/christopher-s-jones/awesome-command-line]. The entire book source code is released as an open source project under the Creative Commons Attribution-ShareAlike 4.0 International license, and may be used under those terms.

[1] UNIX is a registered trademark of The Open Group. Linux is the registered trademark of Linus Torvalds in the U.S. and other countries. Apple, Mac, and macOS are trademarks of Apple Inc., registered in the U.S. and other countries and regions. Microsoft and Microsoft Windows are trademarks of the Microsoft group of companies. References to companies or trademarked products within this work are provided for informational purposes only and do not imply endorsement, sponsorship, or affiliation by those companies. The inclusion of such references is solely for the purpose of illustration or commentary, and the author and publisher do not intend to suggest any formal relationship or approval by the respective trademark holders.

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

The Command Line Is For Everyone

Getting Started

The *command line* is a tool of empowerment—a magic portal to your computer that helps you navigate, organize, create, and refine your files on your computer. It also helps you automate and manage tasks running on your computer. When repeatedly clicking and choosing with your mouse becomes tedious, the command line *prompt* allows you to free yourself and just issue a powerful command that does all that you want, quickly and with ease, so you can get back to the creative side of your work. You can turn complex tasks that require a series of many steps into one workflow that accomplishes what was once inconvenient. Each of the major computer operating systems—including macOS, Linux, and Windows—all provide ways to use the command line[1]. While it is an extremely helpful tool on your local desktop or laptop machine, the command line is essential for running remote services, such as a website. This idea in itself is magical! The Internet predominantly runs on Linux-based servers in remote data centers around the world, and access to these servers largely happens through the command line. Can you imagine that? Managing computers remotely around the world? Yes! Thank you command line! We will be focusing on using the command line locally on your computer and how it can be a boost in your workflow, while laying the groundwork for you to expand your computing skills even further.

This book serves as an introduction to the command line and how to get started with it. We first get set up with the amazing tool you need to use the command line—a *terminal* application that knows how to handle your commands. But wait, you say—What is a *terminal*? What is a *command*? What is a *prompt*? In this chapter we introduce these and other concepts in a step by step manner so you have a solid understanding of the terms and lingo needed to use this magic little gem that may be unfamiliar now, but will soon be your trusty friend at your side. In the subsequent chapters we dive in with hands-on learning by highlighting twenty-one essential commands that build your command line foundation. Take a look at *Table 1*! It shows our road map of the commands we cover in each chapter.

Table 1. The 21 commands covered in the book, by chapter and category.

Chapter 2 Core Commands	Chapter 3 File Commands	Chapter 4 Folder Commands	Chapter 5 Text Data Commands	Chapter 6 Utility Commands
`pwd`	`echo`	`mkdir`	`cat`	`less`
`man`	`mv`	`rmdir`	`sort`	`history`
`clear`	`cp`	`du`	`head`	`open`
`cd`	`rm`		`tail`	
`ls`	`nano`		`grep`	

Each chapter builds on the previous. In addition to each command, we learn key concepts along the way that make using the command line progressively more powerful. These concepts include *file and directory paths*, *tab completion*, *redirection*, *expansion*, and *pipes*. Before we get into the details, let's first take a step back and think about the metaphors we use to interact with our computers.

The graphical desktop metaphor

When we turn on our computers, we are greeted with beautiful desktop images, icons, windows, menus, a pointer, and other *graphical* symbols that help us navigate our system and enable our digital creativity. Collectively, these components are known as the *graphical user interface*, or *GUI* for short. We predominantly work with our computers in this manner, and for good reason—it works really well! So think of the graphical user interface as an analogy, or metaphor, that helps us work with the underlying hardware of our machines—processors, storage drives, memory, etc. For instance, a *folder* icon on your Desktop represents a collection of *files*, and a *file* icon represents some data stored on your computer's drive. These metaphors are highly intuitive, but there are times when graphical tools slow us down—usually when you need to work on something repeatedly, remotely, when working with complex or large amounts of data, or when there is currently no way to do what you want using the graphical interface.

The command line metaphor

A *command line interface*, or *CLI*, is also a way to work with your computer by means of a metaphor, but in this case we use plain text words and other combinations of characters as symbols to indicate to the computer what you want to accomplish. Sometimes they resemble English words, other times they are shortened versions of words, or acronyms of multiple words. For example, a command that we cover in Chapter 2 is `pwd`. This is just a three-character command that stands for "print working

directory". Here's a quick example:

```
pwd           <-- This is the command
/Users/chris  <-- This is the result of the command
```

What a little gem! We'll go into the details of this command later, but you can see that it is a very simple command that shows you what directory (i.e. folder) you are currently working in by dispaying the `/Users/chris` text below the command. So the combination of those three letters, when typed in an application that knows how to handle them (a terminal), will give you back a result that shows your current folder, all using text-based symbols.

In the remainder of Chapter 1, we will get started with setting up a terminal application that provides a command line interface for your operating system (macOS, Linux, or Windows). Once complete, we will open the terminal application, adjust the font family and size so that it is comfortably readable, and then introduce the various components—the *shell*, the *command line*, the *command prompt*, and the various parts of a *command*. Let's get rolling!

Terminal Applications

A computer *terminal* is merely a way to send input to a computer and receive the output results. You might also hear the word *console* used interchangeably with terminal, but the latter is a bit more specific in that it was historically a screen device and keyboard connected directly to a mainframe computer and used by the operators. Terminals, on the other hand, were connected to computers remotely on a network. We now of course have terminal applications that are considered *virtual terminals* that emulate these older physical terminals. See *A History of Modern Computing* (2003) by Paul E. Ceruzzi [2] for more background. There are many alternative terminal applications to choose from, but we will start with the default applications on each operating system in order to get set up. For Mac, the default application is called *Terminal.app*, and on Windows 11 we will focus on the default *Windows Terminal*. If you are using Linux, there are many distributions available, but we will focus on Ubuntu®[3] 24.04 and the *Gnome Terminal* that comes packaged with it. Skip to the next section that is pertinent to you for your operating system and we'll get started with a terminal application!

- Setup for Mac
- Setup for Linux
- Setup for Windows

Setup for Mac

If you are on a Mac, Apple has included the Terminal.app since it introduced Mac OS X in 2001, so it has had many years of refinement. You can search for "Terminal.app" using Apple's Spotlight search by pressing the `Command`+`Spacebar` keys at the same time and typing "terminal" (without the quotes) in the search bar. Alternatively you can open a *Finder* window and navigate to the *Applications* > *Utilities* folder to find the Terminal application, as shown in *Figure 1*. Double click the icon to open the application.

Figure 1. Open the Terminal.app in the Applications > Utilities folder.

Adjust the font size

That's it! You should see a window open similar to *Figure 2*, althought the default color may be different based on the Appearance settings on your Mac.

Figure 2. A terminal window example on a Mac.

If you are in Dark Mode, you'll likely see a dark window, and in Light Mode you should see a light window. In your Dock at the bottom of your screen, you can press `Control` + *click* on the Terminal icon (or use your *secondary-click* on your mouse) to bring up the icon menu, and choose *Options* > *Keep in Dock* to add it permanently to your Dock.

To finalize your setup, adjust the font size in your terminal so that you can comfortably see the text. You can also change the font family, but be sure to use a *fixed width* font since the terminal expects it for layout purposes. In order to change the font size, select the *Terminal* menu item and choose the *Settings ...* item. In the Terminal.app Settings window, select the *Basic* (Default) Profile, and the *Text* tab in the window panel. Use the *Change ...* button to change the font size, as shown in *Figure 3*.

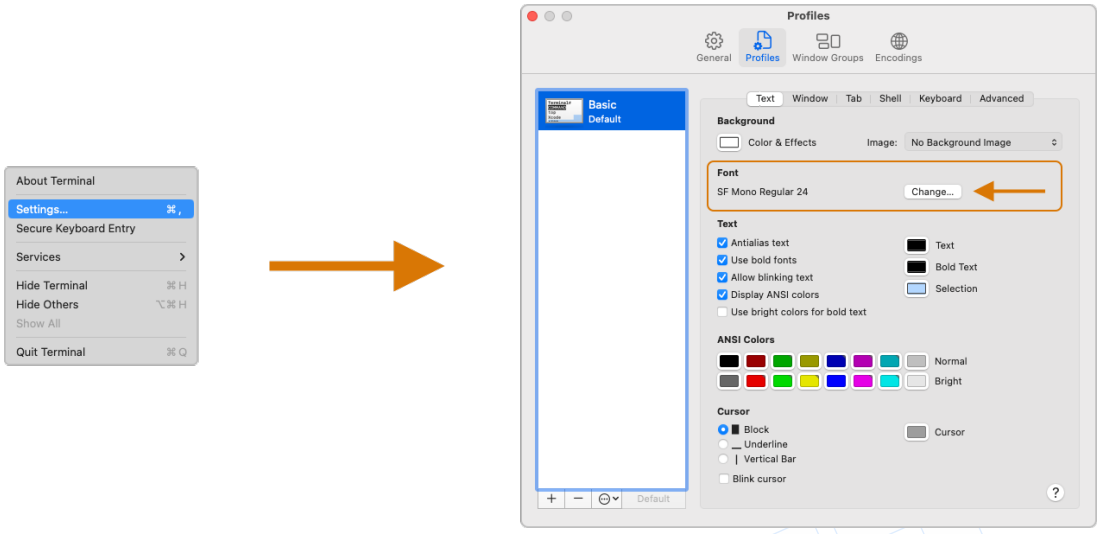

Figure 3. Use the File > Settings... menu item to change the font size as needed in the Terminal.app settings.

You are all set! It's that simple to get configured to use the command line on a Mac. You can continue on to the section entitled *The Shell* to become more acquainted with the command line. Thank you for focusing your power on the magic of the command line!

Setup for Linux

Getting set up on Linux is quite easy as well. On Ubuntu 24.04, the default desktop manager is Gnome. To search for applications, similar to Apple's Spotlight function, press the `Super` key next to the `Alt` key on your keyboard.

If you are on a Windows-branded machine, the `Super` key may have the Windows logo on it. It's also called the `System` key. If you have Linux installed on Mac hardware, this is the `Command` key.

In the search box, type "terminal" (without the quotes), and the default Terminal application icon should be in view. Click on that icon to open the application. You're all set! Once open, you may want to right click on the icon in the *Dash* (i.e. the Application Dock), and choose the *Pin to Dash* menu item so that you have quick access to the Terminal application. See *Figure 4* showing how to search for applications on the Ubuntu Linux Desktop.

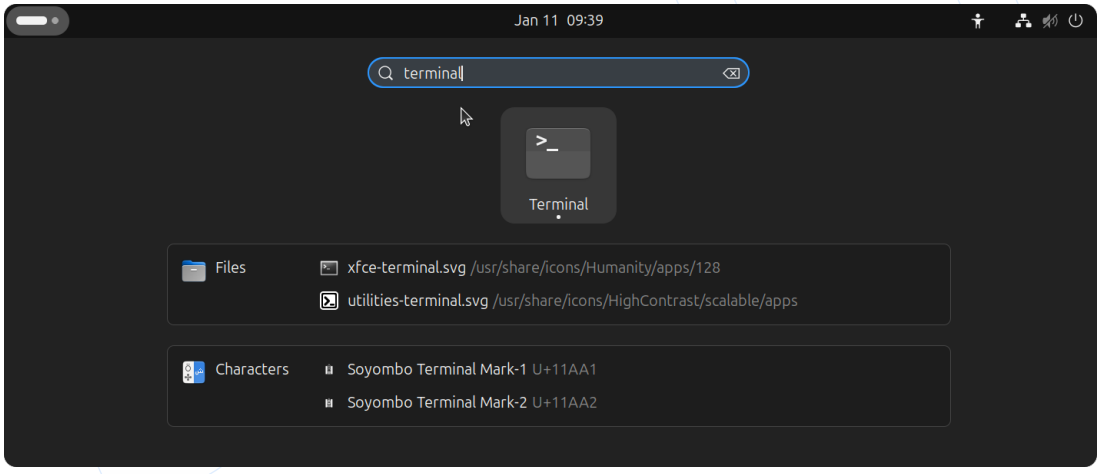

Figure 4. Search for the Terminal application on Ubuntu Linux.

Great! Now that you have the Terminal application running, you should see a window similar to *Figure 5*. Your colors may be different depending on your Appearance settings, but you will either see a Light Mode or Dark Mode window.

Figure 5. A terminal window example on Ubuntu Linux.

Adjust the font size

To finalize your setup, adjust the font size in your terminal so that you can comfortably see the text. You can also change the font family, but be sure to use a *fixed width* font since the terminal expects it for layout purposes. In order to change the font size, select the menu button in the top window bar and choose the *Preferences* item. In the Terminal Preferences window, select the *Unnamed* (Default) Profile, and the *Text* tab in the window panel. Use the *Custom font* checkbox and then the font button to change the font size, as shown in *Figure 6.*

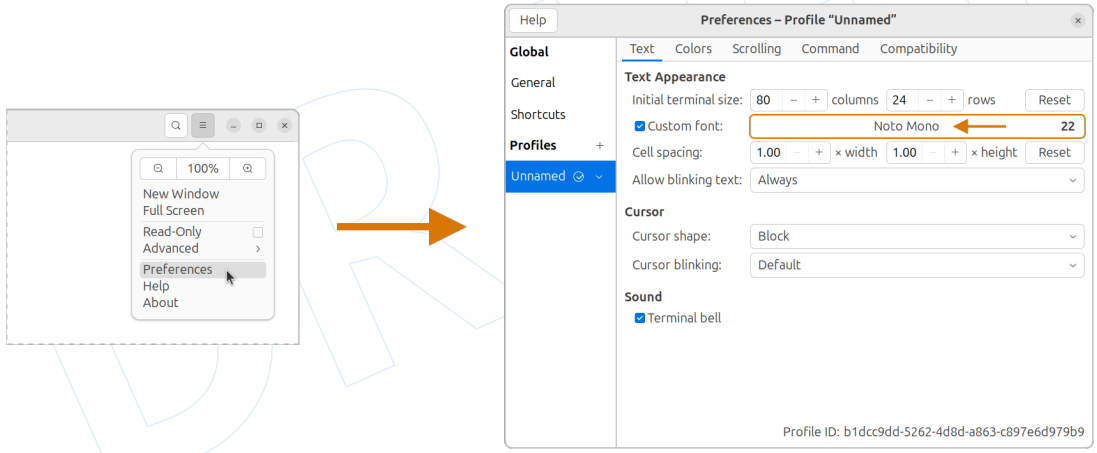

Figure 6. Change the font size as needed in the Terminal preferences.

That's it! It's that simple to get set up to use the command line on Ubuntu Linux. You can continue on to the section entitled *The Shell* onto become more acquainted with the command line. Thank you for taking the next step as a command line magician!

Setup for Windows

The Microsoft Windows operating system has a rich history, but one that is diiferent from the Unix-like operating systems of macOS and Linux. Because of the low-level differences in the systems, Microsoft has created a component called the *Windows Subsytem for Linux*, otherwise known as *WSL*. WSL provides those of us using Windows an integrated system with a full Linux command line environment. In this section, we will complete the following list:

1. Open the Windows Terminal application as an Administrator.
2. Install the Windows Subsystem for Linux component.
    - Install a distribution of Ubuntu Linux.
    - Restart the computer.
3. Enable the Windows Subsystem for Linux required features.
    - Restart the computer.
4. Set up Ubuntu Linux in Windows Terminal
    - Open the Windows Terminal application.
    - Open an Ubuntu Linux tab.
    - Create a Linux user and password.
5. Adjust the terminal font size as needed.

After the Windows Subsystem for Linux installation, the Windows Terminal application will have built-in support and integration for WSL, and will give you a full Linux environment to work with. So let's get started!

Open the Windows Terminal application

Windows Subsystem for Linux is considered a developer tool, and as such, the recommended way to install it is by issuing a command in the terminal application as an Administrator of the computer. To get started, click on the Windows Start menu icon in the Windows Taskbar, or press the `Super` key on your keyboard.

As mentioned before, the `Super` key may have the Windows logo on it, and is usually

next to the `Alt` key.

In the search bar, type "Terminal" (without the quotes). You should see a search result with the Windows Terminal icon. As shown in *Figure 7*, choose the *Run as Administrator* option in the details pane for the Terminal application.

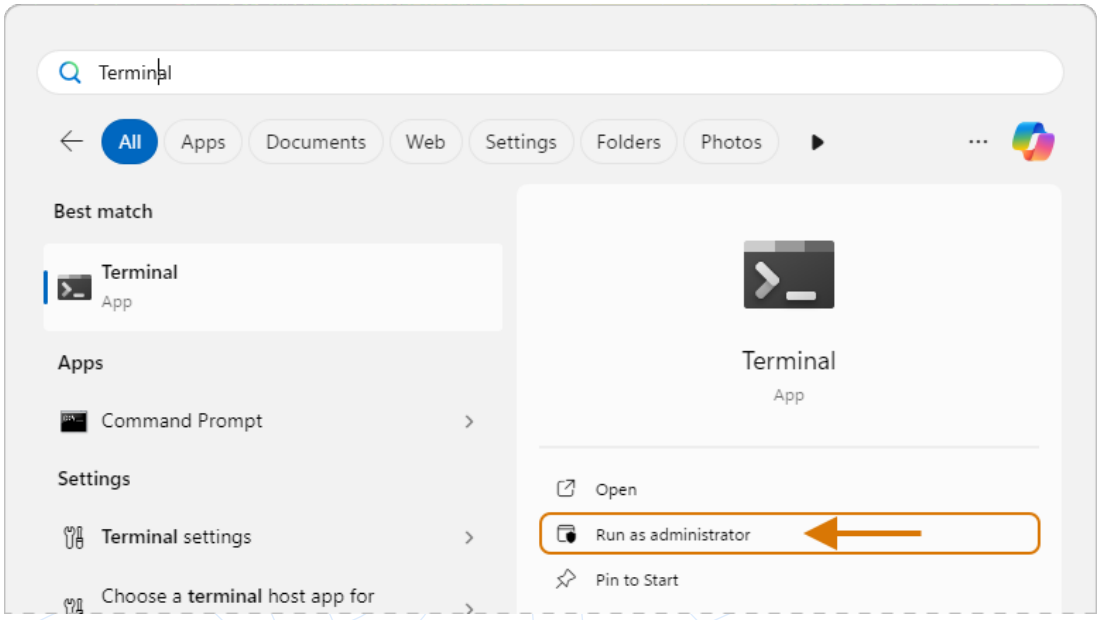

Figure 7. Search for Windows Terminal application and run it as an administrator.

When run as an Administrator, you will see a dialog asking you to make changes to your system, so be sure to choose "Yes" to continue. A terminal window should open and look similar to the window in Figure 8, although the colors may be different depending on your Appearance settings. The Terminal "Powershell" profile usually defaults to a dark background color. To keep this application readily available, *right-click* on the Windows Terminal icon you see in the taskbar, and choose the *Pin to taskbar* menu item.

Install Windows Subsystem for Linux

To install WSL using Windows Terminal, click inside the terminal window and type `wsl --install`, where there is a single space between the `wsl` and the `--install` parts, and press the `Return` key, as shown in *Figure 8*. By running this command, Windows will first download the latest version of the Windows Subsystem for Linux component, and will install the component. It will also install files that are part of the Virtual Machine Platform component that WSL needs for integrating

with the operating system. Once finished, it will prompt you to restart your machine, so do that now.

Figure 8. Run the `wsl --install` *command in the Windows Terminal application.*

Enable the Windows Subsystem for Linux required features

Once rebooted, you will need to ensure that the WSL components are enabled. To do so, click on the Windows Start menu icon in the Windows Taskbar, or press the `Super` key on your keyboard. In the search bar, type "Turn Windows features" (without the quotes). As shown in *Figure 9*, you should see a search result with a Control Panel option for "Turn Windows features on or off". Click on this option to open the features dialog, and scroll down in the dialog toward the bottom.

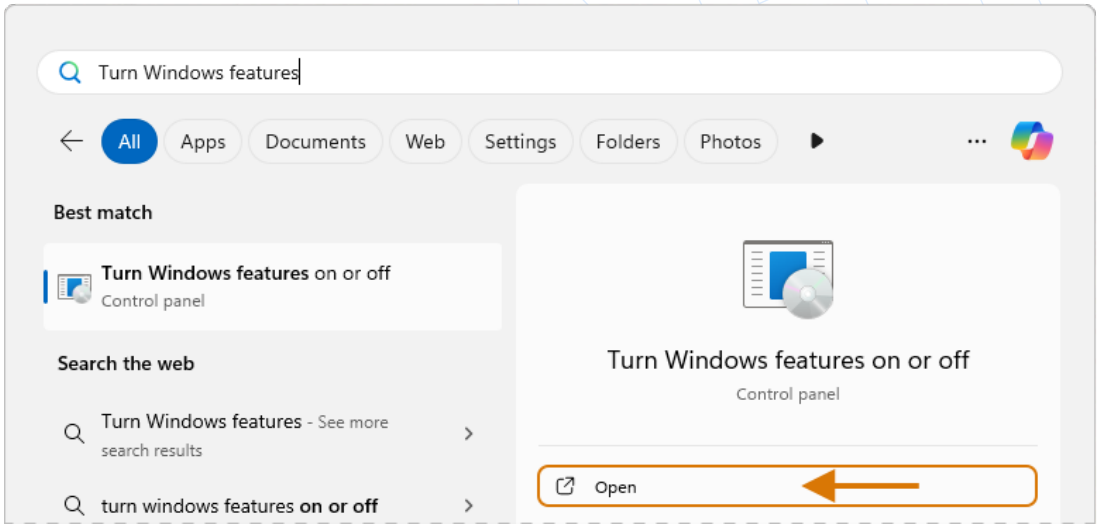

Figure 9. Use Windows Search to open the 'Turn Windows Features on or off' Control Panel.

As shown in *Figure 10*, ensure that the "Virtual Machine Platform" and the "Windows Subsystem for Linux" items are checked. After closing this dialog box, Windows will enable these components, and will prompt you to restart your machine.

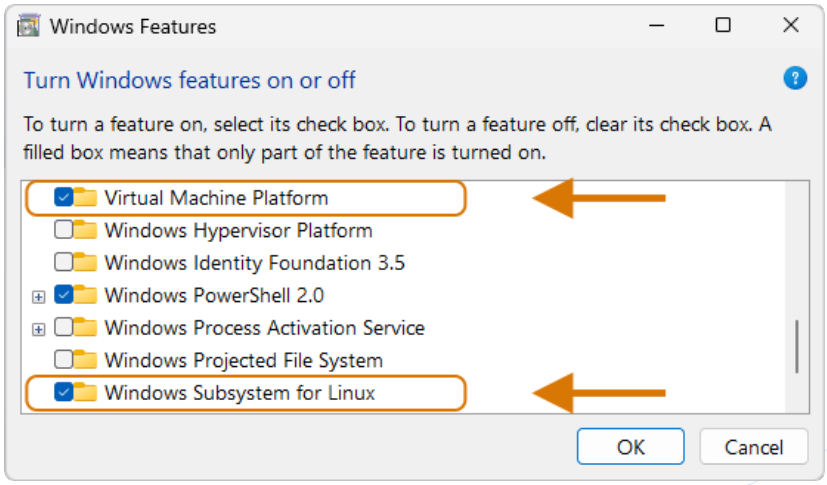

Figure 10. Enable the Virtual Machine Platform and Windows Subsystem for Linux components in the Control Panel.

Set up Ubuntu Linux in Windows Terminal

Great, the underlying components are now installed! It's now time to set up Ubuntu Linux using the Windows Terminal application. So, open the Windows Terminal application again, either from your taskbar or the Windows Start menu. By default, it will open with a Windows PowerShell profile tab. As shown in *Figure 11*, click on the down-arrow icon next to the '+' icon at the top of the window to open a new tab, and select the Ubuntu profile item.

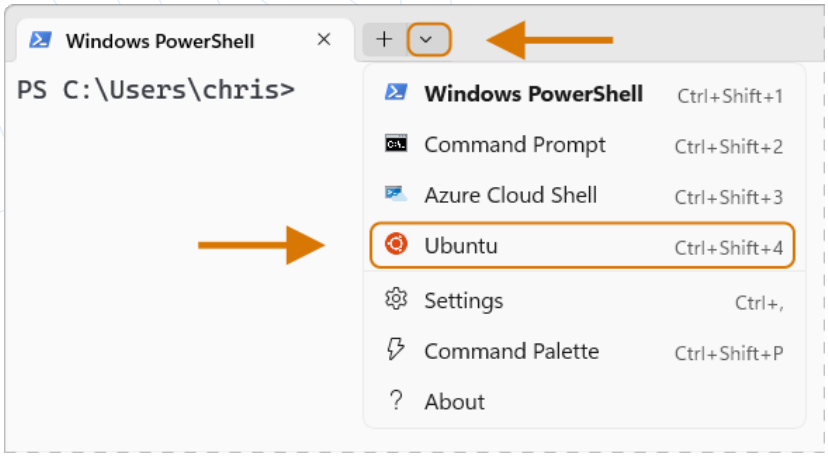

Figure 11. Open an Ubuntu Linux profile using the drop-down icon in the Windows Terminal tab bar (next to the + sign.)

This will initiate the Windows Subsystem for Linux, and will start Ubuntu Linux. It will take a few minutes to initialize, but will then prompt you to create a UNIX username (i.e. Linux username). You can use the same name as your Windows user name, or a different one. After entering your name, and pressing the `Return` key, it will then prompt you for a password. Type in a password of your choosing, and also write it down.

As you type in the password field, your typing will not be visible, which is typical behavior for command line password entry.

Confirm your password a second time when prompted, and your Linux environment will be set up for you! Once the text has stopped scrolling in the window, you will have a fully-functional Linux command line, similar to what is shown in *Figure 12*.

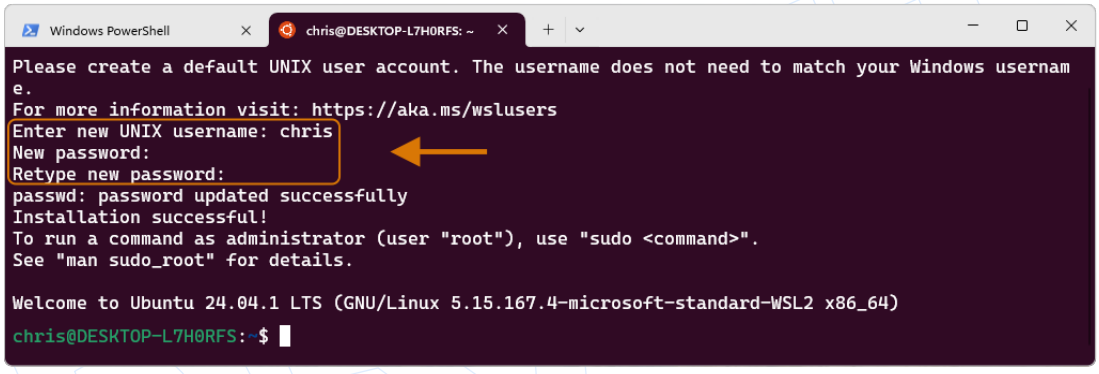

Figure 12. A complete Linux command line running within Windows.

Adjust the font size

To finalize your setup, adjust the font size in your terminal so that you can comfortably see the text. You can also change the font family, but be sure to use a fixed width font since the terminal expects it for layout purposes. In order to change the font size, click on the drop-down icon in the tab bar again, and choose the *Settings* item in the menu. This opens a new tab in the Windows Terminal with the settings for the application, and the settings for each profile, including the Ubuntu profile. In the sidebar on the left, scroll down and click on the Ubuntu profile, as shown in *Figure 13*. The Ubuntu profile settings will appear in the right window pane. Scroll down in this pane, and choose the *Appearance* section.

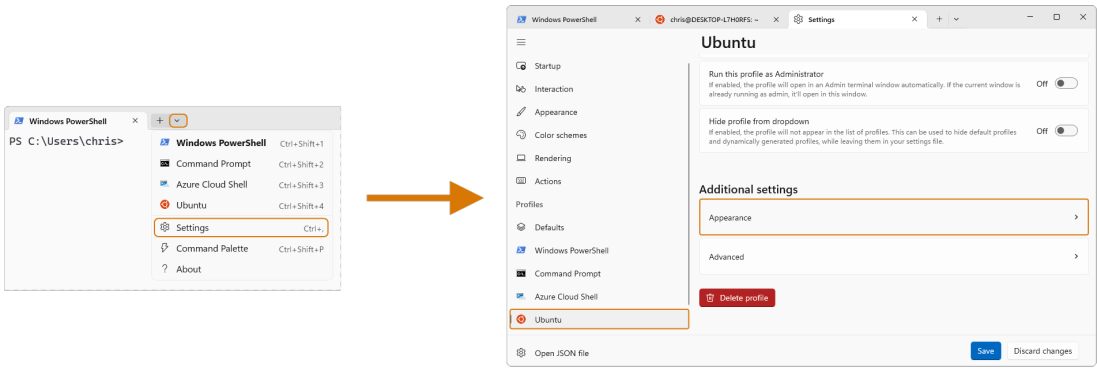

Figure 13. To change the font size, first open the Terminal Settings and choose the Ubuntu profile's Appearance section.

This opens a dialog that allows you to change the font size as needed. See the example in *Figure 14* for changing the font size. Once finished, close the Appearance dialog and click the *Save* button at the bottom of the Settings tab, as shown in *Figure 14*, and then close the Settings tab.

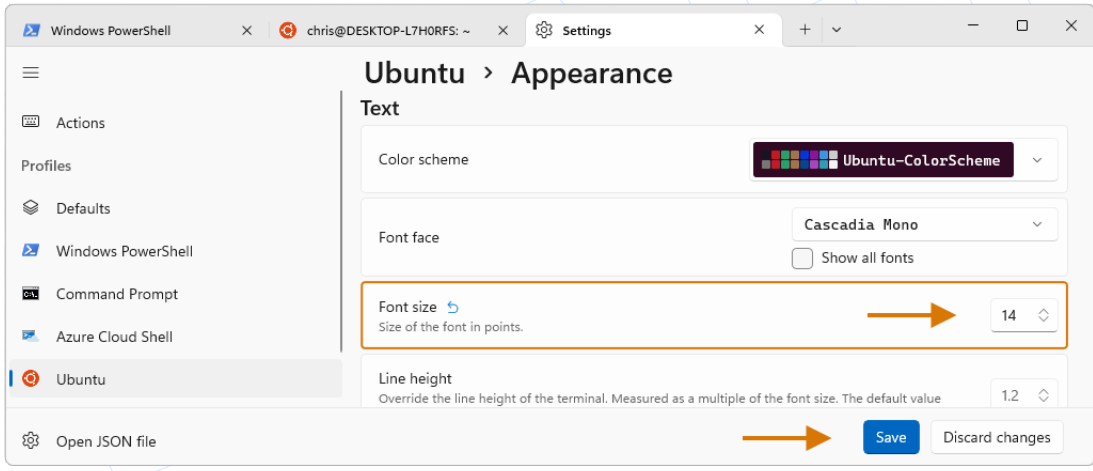

Figure 14. Adjust the font size as needed, and click on the Save button to save the profile changes.

Congratulations! You are ready to continue with your command line journey in the next section to learn about the concept of *The Shell*! Thank you for building your magic command line skills!

The Shell

Now that you have set up a working terminal application, you are well on your way to using the command line with ease! To help with some of the terminology, let's first discuss what a *shell* is. In the course of your work, someone may say "Open up a terminal", "Open up a console", or "Open up a shell". As we mentioned before, these terms are often used interchangeably. However, let's touch on the idea of a shell in more detail.

When you open your terminal application, a number of things happen in the background to set up your environment, such as loading your default settings profile. As part of this process, the terminal will start another process called a *shell interpreter*—which is a program running invisibly in the background—that is waiting for your command to be typed. When you do type the command and hit the `Return` key, the shell program kicks into gear, interprets all of the text that you entered, and runs the command like a programming language. In fact, you are actually writing commands in what is called a *shell language*!

Here's the same example as *Example 1*, but with a comment added to the command:

Example 1: Issuing the `pwd` *command with a comment*

```
chris@ophir ~ % pwd # Issue the pwd command
/Users/chris
```

Notice that the `pwd` characters are followed by a space, then a `#` (hashtag) symbol, and then another space and the comment sentence. The shell interpreter evaluates everything in the command, and validates it based on the shell language rules. In this case, we just learned that you can issue a command, followed by a `#` (hashtag) character and any other written comment, and the shell will ignore any characters to the right of the hashtag because it knows it is a comment, and will proceed to give you back a result.

The take home message here is that the shell interpreter is doing the heavy lifting behind the scenes, and there are many variants of these interpreters. The earliest shell interpreter is attributed to Louis Pouzin in 1964 for the CTSS/Multics operating system.[4] Since 1979 the UNIX operating system included the default shell interpreter called `sh`, and a free version of it is still the default on Linux and Mac. That said, there has been immense improvements to shell programming languages since the 1970s, and many different interpreters, with new features, have been written and shipped with various operating systems. To name a few, there is `ksh`, `csh`, `bash`, and `zsh`.[5] On modern versions of Linux, the default shell tends to be `bash`, and on a Mac it is now `zsh`. For the purposes of this handbook,

we'll see that these shells all work similarly if not identically in some cases. In the next section, we'll take a closer look at the *command prompt*, but know that the shell interpreter is the workhorse behind your magic commands!

The Command Prompt

We are now familiar with opening a terminal application, which in turn spins up a shell interpreter to handle your commands behind the scenes. Now let's familiarize ourselves with the idea of the *command prompt*, which is your go-to location for typing in commands. Once your terminal application has opened, you are presented with an almost empty window, with a few characters written at the top. These characters are followed by the *cursor*, which is some sort of flashing—or not flashing—block character, underscore or other inviting symbol that ever so subtly evokes "type here". Collectively, all of these characters are considered the command prompt—dutifully waiting for you to enter a command. See *Figure 15* for a labeled diagram of a typical command line.

Figure 15. A typical command line, with an example of a default `zsh` *command prompt, showing the user name, the computer host name, the current folder (~), and the* `%` *sign, followed by a block cursor.*

The command prompt on modern systems tend to include your user name, followed by an `@` (at) symbol, followed by the network host name of your computer. There is usually some kind of delimiter character (a space or colon), followed by a `~` (tilda) character (which, as we discuss later, represents your home folder). Lastly, you will see either a `$` (dollar sign) character (for `bash` shells) or a `%` (percent) character (for `zsh` shells). Command prompts can be customized to your liking—modern prompts can be very colorful and include a lot of information, or can be bare bones, depending on your preferences. Take a look at *Example 3* for various command line prompt examples.

Example 2: Examples of various command line prompts.

```
chris@ophir ~ % ❶
chris@nuthatch:~$ ❷
root@nuthatch:~# ❸
>_ ❹
# ❺
```

❶ A `zsh` prompt with username, hostname, current folder, a `%` symbol, with a block cursor

❷ A `bash` prompt with username, hostname, current folder, a `$` symbol, with a block cursor

❸ An administrator prompt with username, hostname, current folder, a `#` symbol, with a line cursor

❹ A minimalist prompt with a `>` (chevron) symbol and an `_` (underscore) cursor

❺ A typical root prompt (administrator) with a `#` symbol

What character shows up in the prompt is configurable, and some people prefer having a minimalist prompt with just a `>` (chevron) symbol, with no username or other information. The command prompt tends to be on the very first line of your terminal window. The combination of the command prompt, and this imaginary first line of text at the top of your window, is considered the *command line*. This is your magic portal that gives you superpowers with your computer, which we will see in the following chapters.

On Unix-like operating systems like macOS and Linux, an account for the administrator (also called the super-user, or root), conventionally is denoted by a `#` (hashtag) symbol in the command prompt rather than a `$` (dollar) or `%` (percent) sign, which denote a regular user. This reminds you to be cautious when issuing commands as the administrator.

The Parts of a Command

In the previous sections we've had a brief look at a very simple command called `pwd`, and we will discuss it further in *Chapter 2. Core Commands*. But to learn about the parts of a command, and to get a feel for command line syntax, let's look at an imaginary command called `catdb`, which is shown in *Figure 16*. The command stands for "cat database", and so you could imagine that we have a database of cat information stored within it, and the `catdb` command allows us to work with the database. In fact, one way to work with it is to search the database and filter the results based on some criteria. The command even has some built-in options to return very popular results, like only returning kitten

records, and cute ones at that, given we are in the Internet Age. The command can also save your search records to a file of your choosing, so you can share your kitten pictures and details with friends. So, given our fictitious `catdb` command, let's discuss the parts of a typical command that are shown in *Figure 16*.

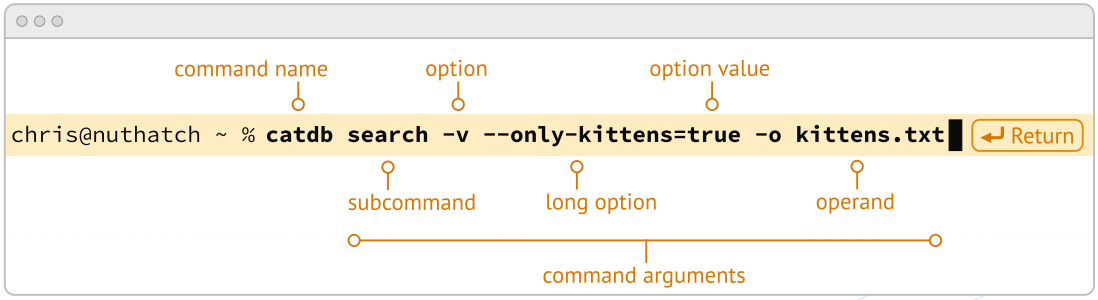

Figure 16. The labeled parts of a command and the command's arguments, including a subcommand, option, long option, option value, and operand.

We start with the *command name* itself, `catdb`. To be able to run this command, it has to be installed on your system, and located in a folder that is well-known to your shell interpreter.[6] Let's assume that our `catdb` command is installed correctly. Next, notice that there is a *space* character after the command itself, and in between the other parts of the full command. This is very important, because the space character acts as a boundary between the command parts, and the shell interpreter will parse the command parts based on these spaces. If you have two or more consecutive spaces between command parts, the shell interpreter treats them as a single space combined, so don't worry about having extra spaces. But yes, be sure to use a space between the parts of a command.

When working with file names that have spaces in the name, use either double-quote or single-quote characters around the file name to tell the shell to treat the spaces as part of the file name. For instance, use "the best cats.txt" or 'kittens are awesome.jpg' if there are spaces in the file name.

After you've typed the command name, you then type the space-separated list of *command arguments*. Command arguments are a way to adjust the behavior of your command, and in the case of our imaginary `catdb` command, we pass in a *subcommand* called `search`, to tell the `catdb` command that we'd like to query the database. The twenty-one fundamental commands we cover in the book don't make use of subcommands, but it's good to know that other commands do use them.

It's important to note that all of the all of the characters we type on the command line

> are case-sensitive, so `catdb search` is all lowercase. Most commands tend to be lowercase, but it's not a steadfast rule. Commands can be created with both uppercase and lowercase, and numbers in them as well.

So after our first command argument called `search`, notice the `-v` argument, which is next in line after the required space character. This is known as a *command option*, which can also be called a *flag*, or a *switch*. Command options are like the knobs or dials on a coffee maker that let you adjust the settings and refine how the coffee turns out. The `-` (dash) character before the `v` is what tells the shell interpreter that this is a command option, and it will treat it as such. In our `catdb` scenario, the `-v` option means that we want it to return *verbose* output, meaning that we want all the cat details we can get from the database. It's very common for commands to have a `-v` option that is a request for verbose output, but note that the `-v` option is command-specific, so it could mean something entirely different when used with a different command. The way to know what options are available for a command is to read the *manual page* that explains how the command works and what it expects. We will cover this topic in *Chapter 2. Core Commands*.

Now we know that you can pass single-letter options to a command, and that the meaning of the option might not be entirely apparent. So a second and more expressive way to modify the command is with *long options*, such as the `--only-kittens=true` command argument in our imaginary scenario. Long options spell out how they modify the command and can be easier to read, but are longer to type. In this case, the long option is `--only-kittens`, and the `--` (double dash) is the indicator to the shell that this is an option. The `=true` portion is setting an *option value*, meaning that the command has a setting of `only-kittens` (for the search), and the value will be set to `true`. So long options are helpful for readability, short options are quick and easy once you are familiar with the command. Both can potentially take option values, but are not required. For instance, the command may set a verbosity level with `-v 8` where the option value could be a number from 1 to 10. Commands often offer both a short option and a long option at the same time. For instance, `-h` and `--help` will often be available and will both print out a short synopsis of how to use the command and what all the options are.

> While we are focused on the short and long option styles, note that you may also see options like `-help` which has the single `-` (dash) of a short option and a full word like a long option. This format is valid as well, but read the manual for the command to know what is expected.

We now come to the last argument of our `catdb` command, which is `-o kittens.txt`. This is a short option that means "write the *output* to the given file name", and so our `catdb` command will

create a file called `kittens.txt` that contains the results of our search, likely with plenty of cat-friendly information. The file name that we pass in is a type of argument called an *operand*, meaning that it is being acted upon in some way by the command, which is the *operator*. Arguments that refer to the output, or results-side of the command are usually considered operands. This is a fine detail, but just know that the terms *arguments* and *operands* are at times used interchangeably.

We have made it to the very end our command, where we see the `Return` key symbol. Commands are executed at the point where you press the `Return` key, so be sure to do so when you've finished writing your command. When you do, in this case, kitten information will be written to the `kittens.txt` file, and you would also normally see additional information printed to your terminal screen in the lines below your command. So that's it! These are the general parts of a command we use on the command line, but what if our command is super long? Will it wrap to the next line? Will it still be readable? Let's discuss those topics.

Single Line and Multi-Lined Commands

A lot of commands can be short and sweet, like the `pwd` command we've seen in the previous sections. But many commands have a lot of options available to modify the command and refine the results that are returned. Some commands include dozens of options, and it may be helpful to use many of them at once. So our command will often not fit on a single line of text available in your terminal window, unless you have a very large screen and can widen the terminal window. So, we'll often see commands wrap to the second and third line of the window, as depicted in *Example 4*.

Example 3: A long command example with many options that wraps to the second line.

```
chris@ophir ~ %  catdb search -v --only-kittens=true --breeds "Maine Coon,
Persian, Siamese, Domestic Shorthair, Bengal" -o kittens.txt
```

Having the command wrap to the next line can work just fine, and will only be executed when you press the `Return` key. But there are times when the command gets very long and complex, and you just want to clean it up. We have the power! You can use a `\` (backslash) character followed by the `Return` key which is used as an *escape character*. The shell interpreter will ignore the `Return` keypress. You can use the `\` (backslash) character as many times as needed to make your single-line command a tidy *multi-line command*, as is shown in *Example 5*.

Example 4: A multi-line command example with options split across lines with a `\` (backslash) character.

```
chris@nuthatch:~$ catdb search -v \
```

```
> --only-kittens=true \
> --breeds "Maine Coon, Persian, Siamese, Domestic Shorthair, Bengal" \
> -o kittens.txt
```

As you type this command, notice that the shell places the `>` (greater-than) character on the following line, indicating that it is waiting for the rest of the command to be typed. Now, when you press the final `Return` key without a `\` (backslash) character, your command will execute. Show all the cats!

When you are working with commands, you will notice that your mouse pointer has no effect on the position of your command line cursor, which takes a little getting used to! For very long commands, either as single line or multi-line commands, there are times when you need to go back and edit a portion of the command that may have been mistyped, or you may want to change an option. You can use your keyboard's `◂` left arrow and `▸` right arrow keys to move the cursor to the left and right, and the `Delete` key will delete characters at the cursor. Take some time to familiarize yourself with moving left and right along your command.

Of course, this can become tedious when you have a very long command and need to edit an option that is close to the beginning of the command, and your cursor is near the end. But wait, there's a handy trick! You can use the `Control`+`a` key combination to skip the cursor to the beginning of your command! To be specific—while holding the `Control` key, also type the `a` key, and zoom—your cursor has raced to the beginning of the command! Likewise, you can use the `Control`+`e` key combination to skip the cursor back to the end of your command. These two keyboard sequences can really speed up your command editing, when your commands get noticeably long.

Some shells also support the `Option`+`◂` left arrow key combination (or `Alt`+`◂`) to move the cursor word-by-word to the left, and the `Option`+`▸` right arrow key combination (or `Alt`+`▸`) to move the cursor word-by-word to the right.

As we type and execute commands with the return key, we inevitably issue a command that wasn't quite what we meant, but it was close! Perhaps there was a single typo in the middle of the command. Instead of re-typing the very long command again, you can use the `▴` up arrow key to scroll up to your previous command, and then edit it. Yes! It's so easy! In fact you can use the `▴` up arrow key multiple times to scroll through your command history, and can use the `▾` down arrow key multiple times to scroll back to your more recent commands. Amazing!

All of these key combinations can be a game changer with command line productivity, so practice using them often, and they will become second nature. With dedication and repetition, using the

command line will become extremely familiar, and you'll notice how rapidly you can get things accomplished without leaving your keyboard. We're just getting started!

Command Line Interfaces are Awesome!

Our computers are wonderful tools for creativity, particularly due to the graphical user interface metaphor that helps us navigate our machines. And now, as we familiarize ourselves with the command line interface, we see that the terminal application can become our trusty friend and a powerful addition to our toolbox. The command line helps us uncover seemingly secret functionality on our computers by using text-based commands to orchestrate our work in a concise and effective manner. In this chapter we have learned how to access a terminal application on Mac, Linux, and Windows. We now have a solid understanding of a shell interpreter that handles the commands we type, what a command prompt is, and how to construct a command with command arguments and the various styles of command options. We now know how to edit single and multi-line commands, and how to move our cursor within our commands with ease. These concepts set the foundation for the upcoming chapters where we learn individual commands that enable us to navigate our computers, create and manage files and folders, and work with our data files in ways that are often impossible with a graphical approach. The command line is truly a tool of empowerment, and a magic portal into your machine. In *Chapter 2. Core Commands*, we will get hands on experience with navigation commands, and will begin to traverse our files and folders with ease, while getting to know the structure of our storage file systems in better detail. Let's go!

[1] Before desktop computing arose, sending commands to a computer was the predominant way to work with them. The success of the UNIX operating system developed by AT&T Bell Laboratories inspired the development of Linux, the architecture of macOS, and later Windows Subsytem for Linux. We focus on commands in these Unix-like systems.

[2] Ceruzzi, Paul E.. A History of Modern Computing. United Kingdom: February, 2003. https://mitpress.mit.edu/9780262532037/a-history-of-modern-computing/

[3] Ubuntu and Canonical are registered trademarks of Canonical Ltd.

[4] See https://multicians.org/shell.html

[5] The Bourne shell (sh) was wriiten by Stephen Bourne at Bell Labs for UNIX and was released in 1979. Also at Bell Labs, David Korn created Korn Shell (ksh) which was released in 1983 for UNIX. An alternative for sh called CShell (csh) was written by Bill Joy at the University of California Berkeley for BSD UNIX, and Brian Fox wrote the Bourne Again Shell (bash), which is an open source rewrite of the Bourne Shell. In 1990, Paul Falstad released zsh as an open source program.

[6] Shell interpreters have a concept of a PATH variable, which contains a list of folders that it will consult in order to find the command you want to run.

Core Commands

pwd, man, clear, cd, ls

Exploring Your File System

In *Chapter 1. The Command Line Is For Everyone*, we set up our terminal application environment, familiarized ourselves with command line concepts and syntax, and now know how to write and edit commands by controlling our cursor. In this chapter, we begin to put these skills to work by using the command line to navigate and view the files and folders that we have on our computers. We are all very familiar with opening a Finder window on a Mac, or an Explorer window on Windows or Linux, and viewing our folders and files. These are the bread-and-butter tools of the graphical user interface. In the command line world, we have commensurate ways to view our folders and files, which we will explore in the following sections. Taken as a whole, the folders and files on your computer's storage drives are organized in a *file system*, which is a means of associating folder and file names with physical hardware locations. For modern *solid state drives* (*SSDs*) and USB sticks, file and folder data are stored on flash-based memory chips, and for older magnetic *hard drives* (*HDDs*), they are stored on spinning magnetic plates. On Windows computers, each physical drive is usually represented by a drive letter like `C:`. A USB drive mounted under Windows may show up as the `D:` drive. Therefore, on Windows, there are multiple top-level drive names, and each have their own file system for organizing folders and files.

On Unix-like operating systems like macOS and Linux, there is a single top-level folder, and all physical drives are represented as a subfolder in this type of file system. The top-level folder is called the *root directory*, and is denoted by a single `/` (forward slash) character.

In computing terminology, a *directory* is synonymous with a *folder*, and it is common to use the word "directory" when using the command line.

When talking about this root directory, it is usually referred to as *slash* (a shorthand for forward slash). *Figure 17* shows the root directory and the immediate sub-directories on a typical Mac installation.

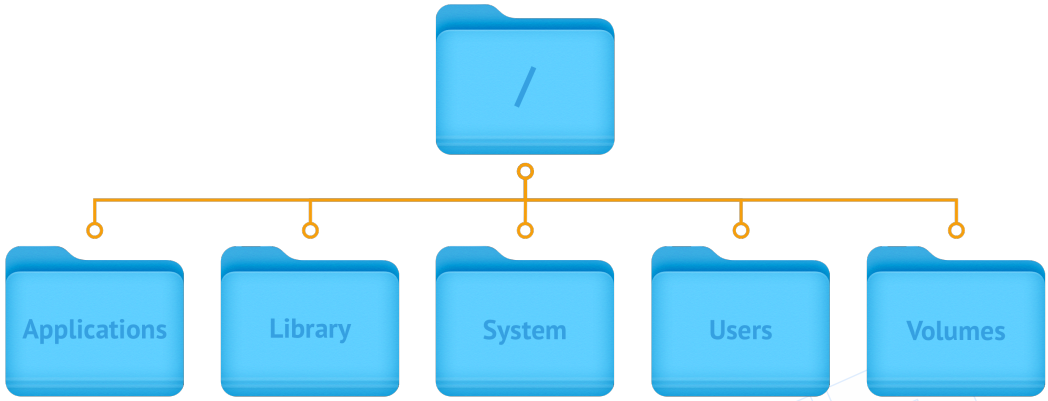

Figure 17. The root directory (called slash) and its immediate sub-directories shown on a Mac.

You can see that everything is organized under a single directory, and we'll be discussing this directory hierarchy in the coming sections. *Figure 17* depicts what first-level sub-directories are shown in the graphical user interface, but in reality there are a number of other sub-directories at this level that can be shown in the command line interface, and we will explore that shortly. But first, let's have a look at our first command that we briefly introduced in previous sections called `pwd`. This command can help us illustrate a location in the file system hierarchy, so let's have a look!

The pwd Command—Print the Working Directory

As we've seen in *Chapter 1. The Command Line Is For Everyone*, opening up a terminal application presents us with a command line prompt in a surprisingly empty window. Let's put that window to use and execute our first command, `pwd`, which is an acronym that stands for "print working directory". In the command line world, the term "print" here means "show the results in the terminal window". Open your terminal and type the `pwd` command, followed by the `Return` key. *Example 6* shows the `pwd` command on a Mac and the result printed on the second line of the terminal.

Example 5: Issuing the `pwd` command on a Mac.

```
chris@ophir ~ % pwd
/Users/chris ❶
```

❶ The result of issuing the `pwd` command is shown on the second line of the terminal.

Awesome! While we've seen this before, this is our first successful command! You can see that the result that printed to the terminal window is, in my case, `/Users/chris`, which should look familiar now that we have a better understanding of the Unix-like file system hierarchy. Your command should show `/Users/<your-username>` if you are on a Mac. This command is telling you that the *working directory*, meaning the current directory location of your command line terminal is your home directory, located at `/Users/<your-username>` in the file system.

And if you are on Linux, you will see a similar result. *Example 7* shows the same command on Linux.

Example 6: Issuing the `pwd` *command on Linux.*

```
chris@nuthatch:~$ pwd
/home/chris
```

Likewise, if you are on Windows with Ubuntu Linux integrated with WSL, you will also see a simlar result. *Example 8* shows the same command on Windows.

Example 7: Issuing the `pwd` *command on Windows using Windows Subsytem for Linux.*

```
chris@DESKTOP-L7H0RFS:~$ pwd
/home/chris (1)
```

(1) Note that the home directory under WSL Linux is different than the Windows home directory of `C:\\Users\chris`, explained later.

We can see some differences across Mac, Linux, and Windows in the results of the `pwd` command, but they all show that the current working directory is the user's home directory. We'll explain the difference between the Windows user home directory and the WSL Linux user's home directly shortly, but suffice it to say that the `pwd` command shows that we all begin our journey on the command line in our account's home folder.

File and Directory Paths

We've been discussing the results of the `pwd` command (e.g. `/Users/chris`), and this is a good time to define what those characters really represent, which is a *directory path*. A directory path is a concise way to represent the path through the hierarchy of the file system tree to a given directory with just text characters. And likewise, a *file path* is a concise representation of a path through the file system tree to a given file. Directory and file paths use the `/` (forward slash) character to separate folder and

file names in the path. This, unsurprisingly, is called the *path separator*. *Figure 18* depicts the directory path represented by `/Users/chris`.

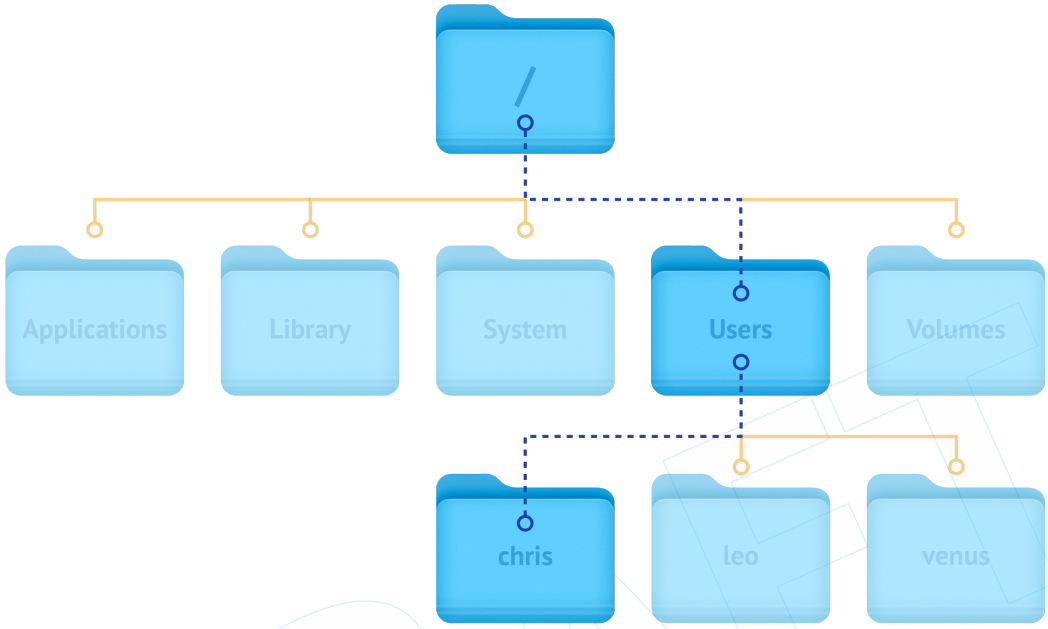

Figure 18. The path from the root directory (slash) to the home directory of the user chris on a Mac, shown as a dashed line.

Now more specifically, the directory path of `/Users/chris` is considered an *absolute path*. It is absolute because it begins with the root directory `/` (slash). On the command line, it can be helpful to refer to files and folders as absolute paths, but it can also be overly verbose, and there is a great shortcut we can use, called a *relative path*. A relative path is the path to a file or directory in the filesystem, but originating from the current directory. But how do we represent what the current directory is, similar to what the `pwd` command produces? Oh, we have the magic! On Unix-like systems, the current directory is represented as a single `.` (dot) character. Wait, what? Just a dot? Yes, surprisingly, just a dot! This is the most succinct way to say "you are here"! To provide an example, let's say that I have downloaded a file from the Internet called `funny-cats.jpg`, and I saved it to my `Downloads` folder in my home directory. On the command line I can refer to this file as a relative path from my current working directory of `/Users/chris` with the following syntax: `./Downloads/funny-cats.jpg`. As we'll see later in *Chapter 3. File Commands*, we can work with files by referring to them with this relative path format. The `.` (dot) character tells the shell interpreter to start from my current directory and traverse into my `Downloads` directory to precisely locate my cat photo. The same file can of course be referenced as an absolute path with `/Users/chris/Downloads/funny-`

`cats.jpg`, but we save some keystrokes by using the relative path format, and when you happen to be located in deeply-nested folders, it can save a lot of typing.

The shell also interprets `Downloads/funny-cats.jpg` as a relative path with no `./` characters at the beginning. Using the `./` (dot slash) syntax is just slightly more explicit.

One last very helpful shorthand symbol for creating relative paths is the `~` (tilda) character. This character indicates your home directory, so if I am logged in as `chris`, `~` is equivalent to `/Users/chris` on a Mac, or `/home/chris` on Linux and Windows WSL. If you want to refer to another user's home directory, you can use `~venus` or `~leo`.

So far we have discussed relative paths from the current directory that are downstream in sub-directories. But what if we want to refer to a file in a directory that is not below our current directory, but up and in another path altogether? Yes, we have the power! By using the syntax of a parent directory, which is represented by `..` (double dot), we can construct a relative path into any other directory in the file system. For example, let's say my teammate, Venus, also has an account on my machine, and has downloaded a really fun cat photo into her `Pictures` directory in her home directory. With her permission, I can refer to this file with the following relative path: `../venus/Pictures/supercat.jpg`. Let's break this down. Since her home directory is `/Users/venus`, I can point to my current parent directory of `/Users` using the `..` syntax, and then into Venus' `Pictures` directory from there. The `..` parent directory syntax is very powerful, because you can chain them together. For instance, from the current directory of `/Users/chris`, I can refer up two parents with `../../`, which is the equivalent of the `/` root directory (up one to `Users`, and up another to `/`).

Table 2 summarizes the common file and directory path components we've discussed in this section.

Table 2. Examples of absolute and relative directory and file paths.

Absolute Paths	Relative Paths
`/home/leo`	`../leo`
`/Users/Pictures/venus`	`./Pictures/supercat.jpg`
`/Users/chris/Downloads/funny-cats.jpg`	`Downloads/funny-cats.jpg`
`/Users/venus`	`~venus`

We'll see more of the relative and absolute paths as we get familiar with more commands. But for now,

let's take a look at how we learn to use any command in the next section on the `man` command!

The man Command—Accessing the Manual

We now have a solid understanding of how to reference file system locations using both absolute and relative paths. Let's turn now to navigating the world of commands, and how to be guided through the details of each command. The creators of these commands really want you to succeed in using them! To facilitate your success, they communicate all of the details about a command in what is called a *manual page*, which contains everything you need to know about a command. When commands are installed on your machine, a manual page also gets installed that provides:

- A short synopsis of how to use the command and the purpose of the command.
- A longer description of the command and how to use it.
- An explanation of each short and long option available for the command.
- Examples of how to use the command.
- Historical information about the authors and other details.

Viewing a manual page

In order to access these manual pages, we will introduce another command, called `man`. Yes, it is shorthand for "manual page"! So when we need to know what a command does, what the options and other arguments are for the command, we use the following syntax: `man <command-name>`, where `<command-name>` is the name of the command that we need guidance on. Let's begin with an example using the `pwd` command, since it is very simple. Go ahead and type `man pwd` and you should see output in your terminal that is similar to *Figure 19*.

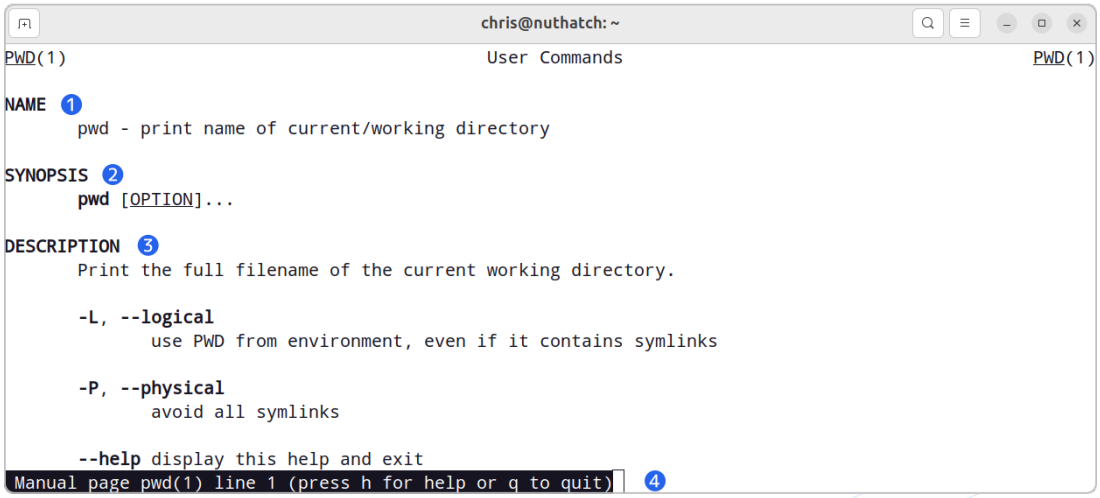

Figure 19. Issuing the `man pwd` *command on Linux.*

Let's discuss the four items highlighted in *Figure 19*.

1 *Name*

You will first see that there is name and a quick definition of the command toward the top of the manual page.

2 *Synopsis*

The synopsis shows the syntax rules for using the command. Square brackets around a command argument means that they are optional, and arguments without square brackets are required. In the case of the `pwd` command, there are no required arguments. The word `OPTION` in this context means any of the short or long options are placed in this position when using the command.

3 *Description*

The description gives all of the details about the command, and will define each of the short and long options that are available.

4 *Paging Information*

Notice that while the output is printed directly in your terminal window based on the size of your window, only a portion of the manual page is shown, discussed below.

Modern versions of the `man` command use a paging mechanism that let's you scroll through the rest of the details, but it is different than scrolling with your mouse. Since manual pages can be very long, navigating them is an art in and of itself, and we will highlight some of the most useful ways to find the

information that you need in the following sections.

If you are on m Mac, the output that you see will be slightly different because the origins of many Mac commands are slightly different than Linux commands.[1]

Moving around in a manual page

Look at the last line of the output in *Figure 9*, which states `Manual page pwd(1) line 1 (press h for help or q to quit)`. This line with the dark background is part of the paging mechanism, and is showing you what line number you are viewing in the manual page. It also lets you know that there is an internal help system to the paging mechanism (by typing an `h` character), and that you can exit the manual page viewer by typing the `q` character (shorthand for quit).

Thank you manual page creators! Let's just summarize a few of the most useful ways to navigate a manual page that are listed in the help section, because there are a lot of key combinations shortcuts listed in the help.

Spacebar

The most direct way to see more of the manual page information is to press the `Spacebar` key, which scrolls through the paged content. This is a quick way to scan through the manual, and it moves you forward one window's worth of the page at a time.

Arrow keys

Likewise, the `▲` (up arrow) and the `▼` (down arrow) keys let you scroll up and down through the window one line at a time to find just what you're looking for.

Quit Viewing

When you are finished reading the manual page, you can use the `q` key to quit the viewer.

Now, there are times when a manual page is very long, and you scroll down through the page to scan for what you are looking for. If you have scrolled past the section you are interested in, how do you scroll back up? The line-by-line `▲` (up arrow) is just too slow—we need to scroll page by page, backward through the manual. With letter keys, we have the power!

Letter keys

- `f`—Scroll forward one window page.
- `b`—Scroll backward one window page.

- `j`—Scroll forward one line.
- `k`—Scroll backward one line.

So the `b` key lets us scroll by page back up! These little gems are right at your fingertips and get you exactly where you want to go in the manual. *Figure 19* shows the useful keys we've highlighted.

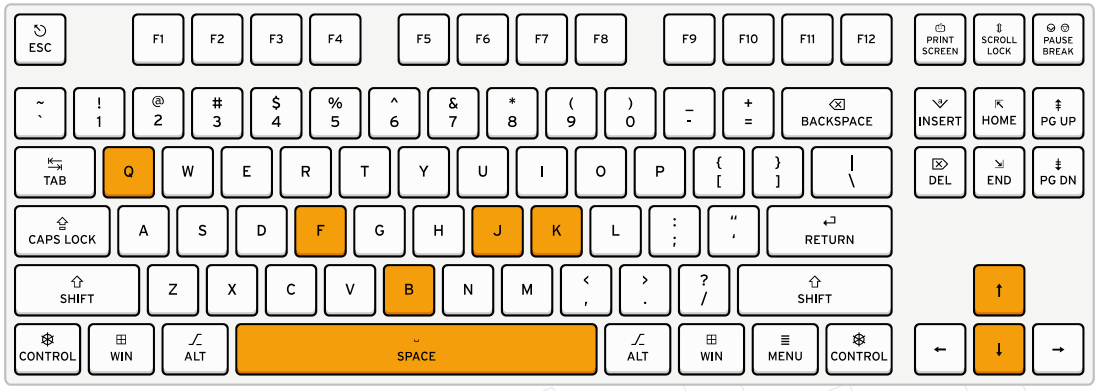

Figure 20. Useful keyboard shortcuts when viewing a manual page, including the spacebar (forward page-by-page); down and up arrows (forward and backward line-by-line); f and b keys (forward and backward page-by-page); j and k (forward and backward line-by-line), and q (quit the viewer).

Great! You now have the tools to navigate any manual page for any command. As you work with commands repeatedly, you will remember many of the short and long options that are available because you use them regularly. Repetition is your friend!

Searching within a manual page

There are some command options that you will use infrequently, so to refresh your memory on how to use them, you can search within the manual page for the exact option you want to use. To do so, use the `/` (slash) key followed by what you want to find.

Let's use the `man pwd` command output as an example again. When you type that command, you will see there is a `-P` short option for the `pwd` command. Let's search for it. Typing the `/` slash key on your keyboard while you are viewing the manual page tells the paging mechanism that you want to search. *Example 10* shows how to search for the `-P` option.

Example 8: While viewing a manual page, a single `/` (slash) key invokes a search, followed by what you want to find.

```
/-P
```

Give it a try! Your cursor will show at the very bottom-left corner of your window. Anything that you type after the `/` (slash) character is considered your *search pattern*. When you press the `Return` key,—Whoosh!—The manual page scrolls directly to the first instance of the `-P` characters. This shortcut can save a lot of time when you know what you are looking for.

Another example would be to search for the EXAMPLES section of the manual page by typing `/EXAMPLES` and the `Return` key. If this section exists for the command it will jump right to it, or it will tell you "Pattern not found".

The navigation keys that are useful when viewing a manual page are derived from the functionality of the `less` command, which we explore in *Chapter 6. Utilities*. So these keyboard shortcuts will come in handy elsewhere.

Occassional mistakes

Okay, we now have a good sense of how to read the manual pages for our commands, and how to navigate the manual page viewer. These skills become second-nature as you practice using commands. It is very common, however, to mistype a command on the command line, and get a very unexpected result! Let's purposefully insert a typo into our command and type `mane pwd`. Try it yourself! *Example 11* shows the output from the shell interpreter.

Example 9: Demonstrating an incorrect command by issuing `mane pwd` on Linux.

```
chris@nuthatch:~$ mane pwd ❶
Command 'mane' not found, did you mean:
  command 'mace' from snap mace (0.2.0)
  command 'mame' from snap mame (mame0270)
  command 'mame' from deb mame (0.261+dfsg.1-1)
  command 'make' from deb make (4.3-4.1build1)
  command 'make' from deb make-guile (4.3-4.1build1)
  command 'mne' from deb python3-mne (1.3.0+dfsg-1)
  command 'man' from deb man-db (2.12.0-1) ❷
See 'snap info <snapname>' for additional versions.
```

❶ Mistakenly typing `mane` instead of `man`

❷ Some helpful information points you to similar command names

Hah! It's easy to make mistakes—they happen all of the time. The shell interpreter prints out a response that lets you know that it didn't recognize the command that you typed, and provides you with a number of possible alternatives that are similar to what you typed. Thanks for tip! Now you can correct your mistake by re-typing the command. That said, sometimes commands can be very long, and re-typing them can be tedious. In the next section where we introduce the `clear` command, we'll also revisit the wonderful shortcut where you can summon a command back like magic!

The clear Command—Keeping It Tidy

In the previous section, we described how to view and navigate a manual page for any command, and when you pressed the `q` key on the keyboard, the contents of the manual page dissapeared. That is because the viewer has built in functionality to clear the screen, which helps you get directly back to your work. However, the output from most of our commands generally stays in the terminal window, and scrolls up and out of view as we type. This is known as your *session history*. Our command prompt always shows back up after the output of the previous command, ready for our next command. But as you can see from our mistakenly-typed `mane` command, the output may be useful in the moment, but it would also be nice to just clear the screen and start anew. Yes, it's so easy! As you probably guessed, the `clear` command does just that—it clears the contents of the terminal window, resets the prompt to the top of the window, and sets us up for our unobstructed next command. Keep it tidy! *Figure 20* shows the results of the `clear` command on a Mac.

Figure 21. Results of using the `clear` command to tidy up your terminal window.

Feel free to type `clear` to clear your terminal window at any time that you feel that things are getting cluttered. When you do so, the command usually clears the visible part of your window. There is also a concept of a *scrollback buffer*, which is the in-memory record of your terminal session from previously typed commands and their output. To scroll back and view your terminal session history, you can use your mouse, trackpad, or mouse wheel. Most terminal applications let you configure the number of lines of scrollback that it maintains in memory so you can scroll back and review or copy any output.

Use the `man clear` command to read the manual page for the `clear` command. There are slight differences between the macOS and Linux versions of the command, but they both clear the active window.

On Mac, you can use the `Command` + `K` key combination to clear the entire scrollback buffer. In Linux and Windows Subsystem for Linux, the `clear` command clears the full buffer, unless you include the `-x` option.

Revisiting the command history

Now that we are able to clear the slate and start with a fresh command prompt at the top of our terminal window, we can re-type our command after making a minor mistake from the previous section when we typed `mane pwd`. But let's assume we issued a very long command that would take a while to type again. Our *command history* is our friend! As we briefly mentioned in *Chapter 1. The Command Line Is For Everyone*, the shell interpreter keeps a history of all of the commands that we run, up to a configurable number of commands. So getting back to them is super easy.

At the command prompt, just press the `▲` (up arrow) key once, and your previous command will show up on the command line. It's like magic! This is one of the most useful shortcuts ever made and is worth repeating here. While using the command line is all about typing, using the modern command line is all about typing the minimum amount to get the job done quickly.

You can now move your cursor left and right to edit you last command, and the `Return` key to re-issue it. So helpful! I'm sure you're wondering about even earlier commands, yes? They are also available! As we've mentioned earlier, pressing the `▲` (up arrow) multiple times will walk you through your command history one command at a time, so you can always get back to your most useful commands. If you pass by a command while arrowing up, you can use the `▼` (down arrow) key to walk forward to your more recent commands. Such a gem!

Now that we are familiar with issuing commands, viewing our current directory, reading the manual pages for commands, and clearing our terminal window, we are now empowered to dive into the two commands that are everyday staples on the command line—the `cd` and the `ls` commands. These two commands are tiny but powerful! Let's learn to travel around the file system and display it all with ease!

The cd Command—Changing Directories

We understand that when we open our terminal application, the shell automatically locates us in our home directory as the starting point. In fact the command prompt tells us this by showing us the `~` (tilda) character, which as we learned is a shorthand for the user's home directory.

In order to move around the file system, we use a very simple command called `cd` which stands for "change directory". It takes one argument—where you want to go! A very simple example would be to change directories to the root directory (`/`) which is the top-level folder. *Example 11* shows us running this command, followed by the `pwd` command to confirm which folder is the current directory.

Example 10: Using the `cd` command to change directories on Linux.

```
chris@nuthatch:~$ cd /
chris@nuthatch:/$ pwd
/
```

Notice that there's no output for the `cd` command, but that the shell has updated the command prompt to reflect our current location, which is now `/` (slash). We confirmed this using the `pwd` command as well. Perfect!

And now, what if we want to return back to the previous directory we were in? The `cd` command has a helpful little shortcut using a single `-` (dash) argument. *Example 12* shows how to return to your previous directory.

Example 11: Using the `cd -` command to toggle back to the previous directory on Linux.

```
chris@nuthatch:~$ cd -  ❶
chris@nuthatch:~$ pwd
/home/chris
```

❶ The `-` (dash) argument means "return to the previous current directory"

This handy little trick can be helpful when you are working in two different directories and want to toggle back and forth between them. Using `cd -` repeatedly will do so. Give it a try!

With no arguments at all, the `cd` command will send you directly back to your home directory. This can be helpful as a reset to get you re-oriented. *Example 13* shows the `cd` command with no argument.

Example 12: Using the `cd` command to return to your home directory on Linux.

```
chris@nuthatch:~$ cd /
chris@nuthatch:/$ pwd
/
chris@nuthatch:/$ cd  (1)
chris@nuthatch:~$ pwd
/home/chris
```

(1) The `cd` command with no argument gets you home

If you are on Windows using Windows Subsystem for Linux, you'll notice that the result of `/home/chris` is different than the Windows user home directory of `C:\\Users\chris`. The WSL Linux user account is different from the Windows user account, but you do have access to all of your files. As we mentioned earlier, Unix-like operating systems have a root directory with everything underneath it, whereas Windows has multiple top-level drives (`C:`, `D:`, etc.) To integrate the Windows filesystem into Linux, the `C:` drive is mapped to `/mnt/c` under Linux, which places it into the single file system hierarchy. But why would it be called `/mnt`? In Unix-like operating systems, external drives and other filesystems are *mounted* to a directory name in order to access it. This is known as a *mount point*. So in Linux, these mount points conventionally are located in the `/mnt` directory.

On Mac and Linux, drives are called *volumes*, and on a Mac they are mounted in the `/Volumes` directory instead of `/mnt`.

As such, the Windows `C:` drive has been mounted into the Linux file system at the `/mnt/c` mount point, and you can access all of your Windows files from your Windows home directory within that drive. So if you are using WSL, go ahead and change directories into your Windows home directory. *Example 14* demonstrates this.

Example 13: In Windows Subsysyem for Linux, changing directories into the Windows user (chris) home directory.

```
chris@DESKTOP-L7H0RFS:~$ cd /mnt/c/Users/chris  (1)
chris@DESKTOP-L7H0RFS:/mnt/c/Users/chris$ pwd
/mnt/c/Users/chris
```

(1) The `/mnt/c/Users/chris` directory is the same as the `C:\\Users\chris` home directory for the Windows chris user account.

Let's next change directories into a sub-directory, and in this case, let's move into the `Pictures` directory in our home directory. *Example 15* shows the syntax for this command.

Example 14: Changing into the Pictures sub-directory from a user home directory in Linux.

```
chris@nuthatch:~$ cd ./Pictures
chris@nuthatch:Pictures$ pwd
/home/chris/Pictures  (1)
```

(1) The current directory is now `~/Pictures`

For Windows WSL users, you'll need to already be in your `/mnt/c/Users/<username>` directory to be able to change directories into `./Pictures` since the WSL installation doesn't automatically add these folders in your Linux home directory in `/home/<username>`.

Tab Completion—Give Your Fingers A Rest

Great! We now understand how to change directories using the `cd` command with both relative and absolute directory paths. Let's now look at a scenario where we have a very deeply nested set of directories within our `Pictures` folder. In our example, I have stored awesome cat and dog photos taken in cities around the world, organized by country, city, and year. Yay pets! But this directory tree is very large, so I would like to traverse it interactively. We can do so using a shell feature called *tab completion*. Tab completion works with the `cd` command. Type `cd` on the command line followed by a space, and then press the `Tab` key twice. If there are sub-directories inside of your current directory, it will list them for you automatically! *Example 16* illustrates this with our pet photo directories from around there world.

Example 15: Using the `cd` command with tab completion to show potential sub-directories to traverse in Linux.

```
chris@nuthatch:~/Pictures$ cd ⇒⇒ (1)
Australia/      France/         Russia/
Canada/         Japan/          United-Kingdom/
China/          New-Zealand/    United-States/
chris@nuthatch:~/Pictures$ cd
```

(1) Pressing the `Tab` key twice shows sub-directories of the current directory path.

Wow! We instantly see what sub-directory choices there are, and because the shell returns our incomplete `cd` command below the list, we can just start typing one of the sub-directory names to add

it to our command. But let's highlight one more feature of tab completion. Let's say from the choices we want to change into the `United-Kingdom` directory. So let's now type the first two characters of that name (just `Un`), and press the `Tab` key again, as shown in *Example 17*.

Example 16: Typing a partial sub-directory name using tab completion in Linux.

```
chris@nuthatch:~/Pictures$ cd ⇒⇒
Australia/      France/         Russia/
Canada/         Japan/          United-Kingdom/
China/          New-Zealand/    United-States/
chris@nuthatch:~/Pictures$ cd Un ⇒ (1)
```

(1) Pressing the `Tab` key once completes the sub-directory to the point where there are multiple matching options.

Ah! So the shell has now filled in the command to be `cd United-` because it knew we wanted to enter a sub-directory that begins with the letters `Un`. But it encountered a fork (between `United-Kingdom` and `United-States`), and has stopped until we give it guidance. We can type the single `K` character that is part of the `United-Kingdom` directory name, and then press `Tab` key again to let the shell auto-complete the directory name, as seen in *Example 18*.

Example 17: Resolving multiple choices in tab completion by providing a unique path direction (the `K` character) in Linux.

```
chris@nuthatch:~/Pictures$ cd ⇒⇒
Australia/    France/         Russia/
Canada/       Japan/          United-Kingdom/
China/        New-Zealand/    United-States/
chris@nuthatch:~/Pictures$ cd United-K ⇒ (1)
chris@nuthatch:~/Pictures$ cd United-Kingdom/
```

(1) Pressing the `Tab` key once completes the sub-directory to the point where there are multiple matching options.

Tab completion is a massive time saver, and it takes just a little practice to consistently use the `Tab` key to let the shell do as much of the typing work as possible. This allows you to drill down into deeply nested folders very quickly, by building a long path on the command line using the tab completion feature. *Example 19* shows how we can use tab completion to build a long directory path interactively.

Example 18: A long directory path built using the tab completion mechanism in Linux.

```
chris@nuthatch:~/Pictures$ cd ⇒⇒  ❶
Australia/      France/         Russia/
Canada/         Japan/          United-Kingdom/
China/          New-Zealand/    United-States/
chris@nuthatch:~/Pictures$ cd United-Kingdom/London/20 ⇒⇒ ❷
2010/ 2012/ 2014/ 2016/ 2018/ 2020/ 2022/ 2024/
2011/ 2013/ 2015/ 2017/ 2019/ 2021/ 2023/ 2025/
chris@nuthatch:~/Pictures$ cd United-Kingdom/London/202 ⇒⇒ ❸
2020/ 2021/ 2022/ 2023/ 2024/ 2025/
chris@nuthatch:~/Pictures$ cd United-Kingdom/London/2024/Awesome- ⇒⇒ ❹
Awesome-Cats/ Awesome-Dogs/
chris@nuthatch:~/Pictures$ cd United-Kingdom/London/2024/Awesome-Cats/
chris@nuthatch:~/Pictures/United-Kingdom/London/2024/Awesome-Cats$ pwd
/home/chris/Pictures/United-Kingdom/London/2024/Awesome-Cats
```

❶ Use double `Tab` keys to interactively see sub-directory options for countries

❷ Do it again when there are still multiple sub-directory options for years

❸ And again for years in the 2020s

❹ And one last time to find cats versus dog photos

You can see that we can use the `Tab` key to quickly build the relative directory path in the command `cd United-Kingdom/London/2024/Awesome-Cats/` . When we finally press the `Return` key, we are whooshed into that directory.

This rounds out our tour of the `cd` command and the magic tab completion feature that makes you a turbo-typer. Go ahead and practice these techniques in the folders of your home directory to get a solid feel for changing directories. Next, we have a look at ways to view file and directory information in more detail using the `ls` command. See you there!

The ls Command—Listing Files and Folders

In the previous section, we have seen that it is very easy to move anywhere within the file system on the command line. In fact, if you need to get somewhere quickly, the tab completion feature will get you there the fastest. But once we have arrived at a given directory, we certainly want to know what files and folders are present, and other information about them. When we're using our graphical tools, we will open up a Finder window on a Mac, or a File Explorer on Linux and Windows. These are of course great tools! We're shown either a list view or an icon view of the contents of the directory, along with some item details like modification dates and file sizes. *Figure 22* shows a typical file explorer view in Linux showing the sub-directories and files in the home directory.

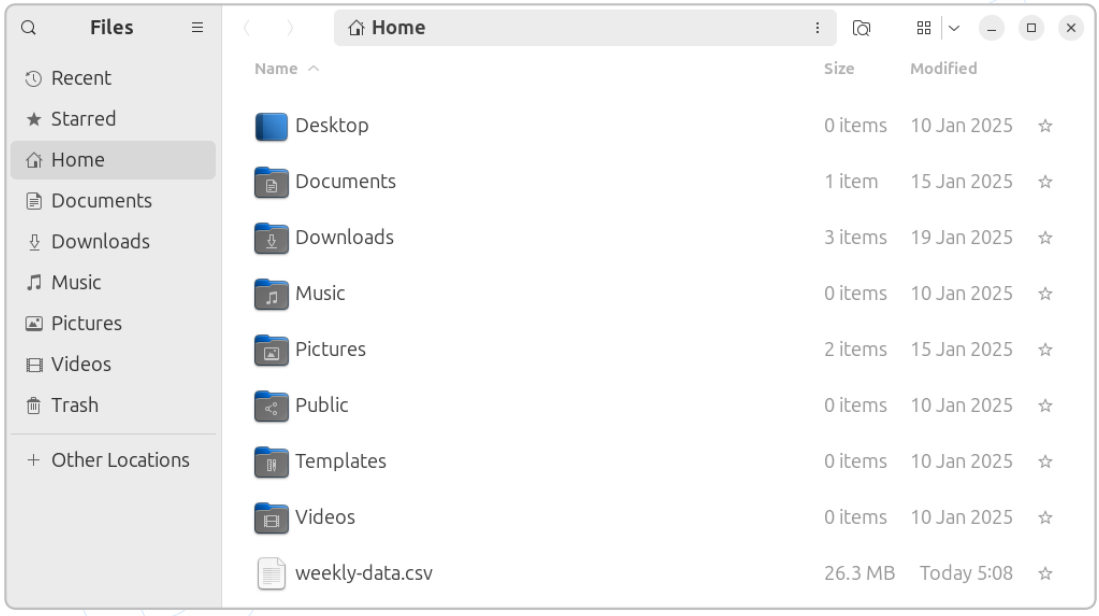

Figure 22. The home folder on Linux showing the sub-folders and files.

To list files and folders on the command line in a similar, but very concise fashion, we use the `ls` command, which is a very small but powerful command that means "list directory contents". We will start with the simplest use of the command, which is to issue it without any arguments. *Example 20* shows the results.

Example 19: Using the `ls` command to show the contents of the home folder on Linux.

```
chris@nuthatch:~$ ls
Desktop  Documents  Downloads  Music  Pictures  Public  Templates  Videos
weekly-data.csv
```

Very simple! You'll notice that the directories are listed in alphabetical order horizontally across the terminal window if your window is wide enough to accommodate all of the names on a single row. If not, they are listed in vertical columns alphabetically. In *Example 20*, the item names in the directory wrap to a second line, so your output may look slightly different. Give it a try!

As you can see, an unmodified `ls` command works well for having a quick look at a directory's contents, but when there are dozens or hundreds of items in a directory, it can be a bit unwieldy. To tidy up the output, we can add a `-1` (numeral one) option to the command, which tells `ls` to list the contents in a single column. *Example 21* shows the results of an `ls -1` command.

Example 20: Using the `ls` command to produce a single column listing (`-1`).

```
chris@nuthatch:~$ ls -1 (1)
Desktop
Documents
Downloads
Music
Pictures
Public
Templates
Videos
weekly-data.csv
```

(1) The `-1` option produces a single column of directory items

Excellent! That is very tidy, and gives us an alphabetical listing in a single column. Try this command for yourself as well. The repetition will start to train your muscle-memory!

Now, if you use the `man ls` command to read about the avaiable options, you will notice that there are a lot of options for such a tiny command! *Example 22* shows the manual page synopsis on a Mac.

Example 21: The many options available to the `ls` command on Mac and Linux.

```
chris@ophir ~ % man ls
```

```
LS(1)          General Commands Manual          LS(1)

NAME
     ls - list directory contents

SYNOPSIS
     ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,]
        [--color=when] [-D format] [file ...]
```

We will be highlighting a few of the most useful options for the `ls` command, since it really is indispensible for quickly viewing the contents of your folders. The most common option is the `-l` (lowercase letter l) option, which produces what is called a *long listing* of your directory. It is popular because it packs a lot of critical information into a small space, but when you first look at it, it may seem a bit foreign! So we will learn how to read a long listing in the next section.

How to read a directory long listing

Example 23 shows the results of the `ls -lh` command on Linux, where the `-h` option produces human-readable file sizes. Notice that you can combine short options with a single dash, like `-lh`.

Example 22: Using the `ls` command to produce a long listing (`-l`) with human readable sizes (`-h`) on Linux.

```
chris@nuthatch:~$ ls -lh
total 26M
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Desktop
drwxr-xr-x  2 chris chris 4.0K Jan 15 11:41 Documents
drwxr-xr-x  2 chris chris 4.0K Jan 21 17:03 Downloads
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Music
drwxr-xr-x 11 chris chris 4.0K Jan 22 12:05 Pictures          (1)
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Public
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Templates
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Videos
-rw-rw-r--  1 chris chris  26M Jan 22 05:08 weekly-data.csv
```

(1) The long listing of items in *chris'* home folder with accompanying details

Okay—that's looking packed full! The long listing provides not only a vertical listing of folder and file names that are alphabetically sorted by default, but every line also provides technical details for each item in the list. *Figure 23* explains the output table with each of the columns of detail, and highlights the far right column with the sub-directory and file names.

```
chris@nuthatch:~$ ls -lh
total 26M
```

drwxr-xr-x	2	chris	chris	4.0K	Jan 10 10:55	Desktop
drwxr-xr-x	2	chris	chris	4.0K	Jan 15 11:41	Documents
drwxr-xr-x	2	chris	chris	4.0K	Jan 21 17:03	Downloads
drwxr-xr-x	2	chris	chris	4.0K	Jan 10 10:55	Music
drwxr-xr-x	11	chris	chris	4.0K	Jan 22 12:05	Pictures
drwxr-xr-x	2	chris	chris	4.0K	Jan 10 10:55	Public
drwxr-xr-x	2	chris	chris	4.0K	Jan 10 10:55	Templates
drwxr-xr-x	2	chris	chris	4.0K	Jan 10 10:55	Videos
-rw-rw-r--	1	chris	chris	26M	Jan 22 05:08	weekly-data.csv
❶	❷	❸	❹	❺	❻	❼

Figure 23. Understanding the columns of the long listing output.

❶ The type and permissions of the file or folder

❷ The number of items (called links) for the file or folder

❸ The name of the file or folder's owner (username)

❹ The name of the file or folder's group

❺ The size of the file or folder in bytes (B or K, M, G)

❻ The date and time the file or folder was last modified

❼ The name of the file or folder

Give this command a try for yourself in your home directory. When you're looking at the output, it is helpful to envision it as a table, with the 7th column being the most important (the folder and file names). The 6th column—modification dates—can be very helpful as well when you're interested in when you've last worked on a given file or folder. Likewise, the size of the file or folder in column 5 is useful, and is common in graphical interface listings too, as we see in *Figure 22*. Because we used the `-h` option, file sizes that are normally shown in bytes are converted to Kilobytes (K), Megabytes (M), and Gigabytes (G) to make the large numbers more readable. But what about the first four columns? Let's discuss those.

Because Unix-like operating systems can have multiple user accounts, every file or directory has a set of permissions and a type that are assigned to it, which are shown in column 1. We'll return to those in a moment. Column two is showing how many items are "linked" to the given file or directory in the given row. For instance, the `Pictures` directory has 11 direct items inside of it, which are the country folders for our cat and dog photos!

While the long listing shows 11 items in the Pictures directory, in reality there are 9 sub-directories. The other 2 links are made up of the `.` (dot) item which is the directory itself, and the `..` (double dot) item which is the parent directory. These two hidden directory names are associated with the Pictures folder as well.

To the right of the item count, Column 3 shows that the *chris* user is the owner of each file or directory. Likewise, Column 4 shows a group named *chris* on the system, and all of these files and directories are associated with that group name. But let's now take a look at the most condensed of all of the columns, the permissions and types in Column 1, and how they work in conjunction with the user and group names in Columns 3 and 4.

Understanding file and directory permissions and types

In the long listing output from the `ls -lh` command shown in *Figure 23*, Column 1 shows the file or directory type and permissions for every row of the table. This is packed full of information! Let's break this information down and explain each part so we can read it at quick glance. *Figure 24* highlights the last two rows of the table and shows the type and permission information for the `Videos` directory and the `weekly-data.csv` text file.

First, notice that the information is presented in 10 slots of text characters, where the first slot represents the *type* of the listed item, and the remaining nine slots represent the *permissions* associated with the file or directory item.

Now notice that the `Videos` row has a `d` for the type, which means it is a directory. The `weekly-data.csv` file has a `-` (dash) for the type, which means it is a regular file. These are the most common types you will see.[2]

Let's now look at the next nine slots, which hold information about the permissions for the directory or file in that row. Notice that the permissions are divided into three categories—for the *user* (owner) of the file, for the *group* the file is associated with, and *other* (all other accounts on the system). Packed into each category are four possible permissions[3]— *read*, *write*, *execute*, or *none*—represented by an `r`,

`w`, `x`, or `-` character, respectively.

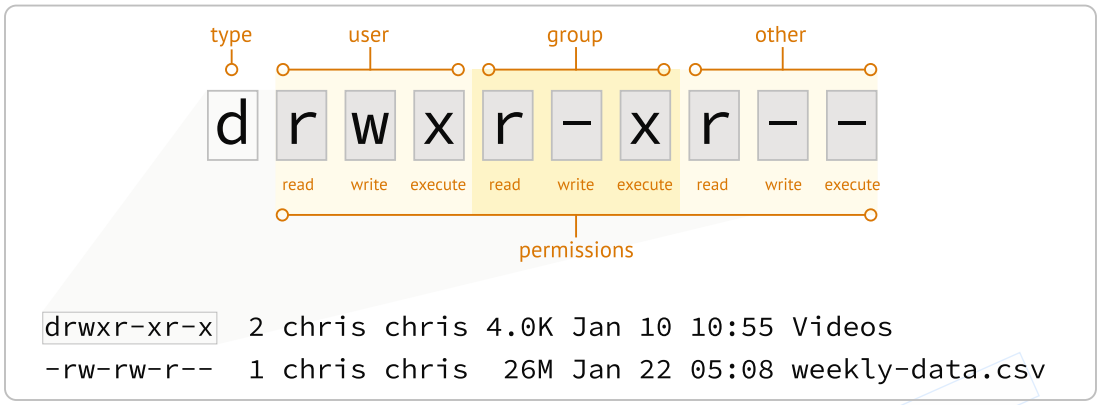

Figure 24. Understanding file and directory permissions and types for `ls -l` command long listings. Examples include the Videos directory and weekly-data.csv file. Permissions for each file or directory are categorized by user, group, and other accounts in the operating system. Read, write, and execute permissions are assigned to each category of accounts.

With this knowledge, we can interpret the permissions for the `Videos` directory and the `weekly-data.csv` file that are shown in *Figure 24* as the following:

Videos directory

- **user**: `rwx` The user *chris* (the owner) can read the contents of the directory, write into the directory (add or change files and folders), and can execute (change into) the directory.
- **group**: `r-x` Anyone in the *chris* group can read the contents of the directory, *cannot* write into the directory (add or change files or folders), and can execute (change into) the directory.
- **other**: `r-x` All *other* accounts on the system can read the contents of the directory, *cannot* write into the directory (add or change files or folders), and *cannot* execute (change into) the directory.[4]

weekly-data.csv file

- **user**: `rw-` The user *chris* (the owner) can read the contents of the file, can write or change the file, and *cannot* execute the file (run it a script or program).
- **group**: `rw-` Anyone in the *chris* group can read the contents of the file, can write or change the file, and *cannot* execute the file (run it a script or program).
- **other**: `r--` All *other* accounts on the system can read the contents of the file, *cannot* write or change the file, and *cannot* execute the file (run it a script or program).

Wow! That is an immense amount of information packed into the long listing output of the `ls -lh` command. While it takes some practice to interpret the file permissions, it does become second-nature. We've discussed four of the most useful options for the `ls` command (`-1`, `-l`, and `-h`), but what if we want to sort the long listing output in ways other than alphabetically ascending? Let's explore those common options next.

Sorting directory listings

Having the long listing output sorted by default in alphabetically ascending order is often exactly what we need. But there are times when the directory has a lot of items, and we want to reverse the direction of the sorting algorithm. We have the magic! We can add a `-r` short option or `--reverse` long option to our command in order to invert the sorting. *Example 24* demonstrates this.

Example 23: Using the `ls` command to produce a reverse-sorted (`-r`) long listing (`-l`) with human readable sizes (`-h`) on Linux.

```
chris@nuthatch:~$ ls -lhr
total 26M
-rw-rw-r--  1 chris chris  26M Jan 22 05:08 weekly-data.csv
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Videos
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Templates
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Public
drwxr-xr-x 11 chris chris 4.0K Jan 22 12:05 Pictures
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Music
drwxr-xr-x  2 chris chris 4.0K Jan 21 17:03 Downloads
drwxr-xr-x  2 chris chris 4.0K Jan 15 11:41 Documents
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Desktop
```

It's that easy. We now have a long listing in alphabetically *descending* order. Let's also try the reverse ordering, but instead of defaulting to an alphabetical sorting, let's sort based on the modification time, using the `-t` option. This is a super useful trick! When you have a lot of files in your working directory, and just want to see what you added or changed most recently when your files scroll by in the listing, you can do a reverse-chronological sorting, as is shown in *Example 25*.

Example 24: Using the `ls` command to produce a chronological (`-t`), reverse-sorted (`-r`) long listing (`-l`) with human readable sizes (`-h`) on Linux.

```
chris@nuthatch:~$ ls -lhrt
total 26M
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Videos
```

```
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Templates
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Public
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Music
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Desktop
drwxr-xr-x  2 chris chris 4.0K Jan 15 11:41 Documents
drwxr-xr-x  2 chris chris 4.0K Jan 21 17:03 Downloads
-rw-rw-r--  1 chris chris  26M Jan 22 05:08 weekly-data.csv
drwxr-xr-x 11 chris chris 4.0K Jan 22 12:05 Pictures
```

So easy! You can see that the `Pictures` directory was the most recently updated and is at the bottom of the long listing output, and the oldest items are at the top. There are many ways to sort the the output of the `ls` command, but one more way to sort that is worth noting is sorting by size. We often accumulate many files in our home and other directories, and it's nice to see them by size, because perhaps we can delete some of the big ones to free up some space! To sort by size, use the `-S` option. Note that it is an uppercase letter `S`! *Example 26* shows the same reverse listing as previously, but sorting by size instad of time.

Example 25: Using the `ls` command to produce a file-size (`-S`) reverse-sorted (`-r`) long listing (`-l`) with human readable sizes (`-h`) on Linux.

```
chris@nuthatch:~$ ls -lhrS
total 3.1G
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Videos
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Templates
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Public
drwxr-xr-x 11 chris chris 4.0K Jan 22 12:05 Pictures
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Music
drwxr-xr-x  2 chris chris 4.0K Jan 21 17:03 Downloads
drwxr-xr-x  2 chris chris 4.0K Jan 15 11:41 Documents
drwxr-xr-x  2 chris chris 4.0K Jan 10 10:55 Desktop
-rw-rw-r--  1 chris chris 4.0M Jan 23 14:54 daily-data.csv
-rw-rw-r--  1 chris chris  26M Jan 22 05:08 weekly-data.csv
-rw-rw-r--  1 chris chris 3.0G Jan 23 14:53 yearly-data.csv
```

You can see that I added a couple of data files into the directory to highlight this point. The files are sorted in ascending order when using the reverse option, and you can see the 3.0 Gigabyte `yearly-data.csv` file at the bottom of the listing.

Viewing hidden files and folders

Unix-like operating systems like Linux and macOS use a convention to hide files from view, which is to begin the filename with a `.` (period, or *dot*). This is very commonly used for configuration files, as we'll see in our directory. You can add the `-a` option to your `ls` command to view your hidden files, as shown in *Example 27*.

Example 26: Viewing hidden files using the `ls -a` *command on Linux.*

```
chris@nuthatch:~$ ls -a
.              daily-data.csv  Music      Videos
..             Desktop         Pictures   .viminfo
.bash_history  Documents       .profile   weekly-data.csv
.bash_logout   Downloads       Public     yearly-data.csv
.bashrc        .gnupg          .ssh
.cache         .lesshst        Templates
.config        .local          .var
```

That's quite a few hidden files and directories! As mentioned above, this is a common way to store configuration data for your applications, and these are collectively know as your *dotfiles*. Notice the `.bash_history` file—this is where the `bash` shell stores your command history, which we have discussed earlier and will explore more in *Chapter 6. Utilities*. There is an equivalent `.zsh_history` file for Mac. Also note the two directory entries named `.` (dot) and `..` (dot dot). We mentioned previously that these represent the current directory and the parent directory, respectively. And there they are, easy to see with a `-a` listing option!

There are obviously many more options to explore with the `ls` command, so go ahead and try them out after reading the manual page using the `man ls` command. As you repeatedly use this command and the `cd` command, you'll find it blazingly fast to find your files. When these commands are combined with the `open` command described in *Chapter 6. Utilities*, you will understand how productive this magic portal can be!

Core Commands Are Awesome!

We've come such a long way in a short period of time when it comes to navigating our file systems! We are now familiar with the concepts of the current working directory, absolute and relative paths, path separators and the idea that Unix-like systems organize everything in a single directory structure beginning with the top-level `/` slash directory. We now know how to use manual pages to read the documentation about any command, and how to navigate within a manual page to find information quickly. It's all coming together! We've also discovered how easy it is to traverse to any location in the file system with the `cd` command. And it's so fast with tab completion! Our command history is at our fingertips, and we can view and sort our files and directories in many different ways—compact, single-column, and long listings. We have a solid understanding of how permissions and types work in a multi-user operating system, and how to view hidden files. The magic is unfolding! These are the foundational commands we will use on a daily basis to work with our files, folders, and data, and let's not forget that we can keep it all tidy with the clear command. So awesome! Keep practicing these commands—it becomes absolutely second nature the more you use them. In the next chapter, we'll be exploring the commands that allow us to quickly create, delete, rename, move, copy, and edit files in our directories, and we will continue to build our foundation with these command line tools and the associated file-handling concepts. See you there!

[1] The origins of macOS stem from the NeXTSTEP operating system (acquired by Apple) and the FreeBSD operating system. The latter is a free and open source version of the Berkeley Software Distribution (BSD) Unix, developed at the University of California Berkeley campus. See https://developer.apple.com/library/archive/documentation/Darwin/Conceptual/KernelProgramming/BSD/BSD.html

[2] There are file types other than regular files and directories in Unix-like operating systems. More information can be found in the manual pages for the `ls` and `chmod` commands.

[3] There are also more possible permissions on Unix-like operating systems. See the manual page for the `chmod` command for more details.

[4] While read permissions set for other accounts indicate they can read contents of the Pictures folder, the permissions set on the parent directory may restrict any access by other users. This is commonly the case for home directory folders.

File Commands

echo, mv, cp, rm, nano

Powerful Ways To Work With Files

In *Chapter 2. Core Commands*, we see how easy it is to change into any directory (with the appropriate permission) and to view our files. So let's now explore how easy it is to work with files directly on the command line. In the coming sections, we'll learn how to use the `echo` command to quickly create a file. This is one of those magical tricks that makes the command line so powerful! We'll then effortlessly append to that newly-created file, and will see just how simple command line file handling is. By highlighting these common command patterns, we'll learn how they are made possible with the concept of *redirection*, and other useful constructs in Unix-like operating systems. We'll also explore how to use the `echo` command to display text directly to the terminal screen, demonstrating the concepts of *standard input*, *output*, and *error*. This let's us then consider the commands for moving and renaming files, copying them, editing them, and of course deleting them. Finally, in order to edit existing files directly in the terminal, we'll also check out a simple command called `nano` that provides us with just the right amount of lightweight editing tools. Let's have a look!

The echo Command—Easily Creating Files

The `echo` command is a simple command which really illustrates the ideas of *inputs* and *outputs* on the command line. We'll discuss why it is called "echo" shortly, but let's first highlight the best part about this command—the fact that you can instantly create a file with it! We'll start with *Example 28*, where we use the `echo` command to write a sentence into a file called `herding-cats.txt` on Windows. If you're on Mac or Linux, give it a try from your home directory too.

Note that creating files with the `echo` command will immediately overwrite a file if it exists, so be careful with this command.

Example 27: Using the `echo` command to create a file on Windows WSL.

```
chris@DESKTOP-L7H0RFS:/mnt/c/Users/chris$ echo Our cat Luna is the ranch
manager. > herding-cats.txt
```

You just created your first real file from the command line! We now have a file called `herding-cats.txt` in our home directory. Let's take a minute to understand what is happening here.

We first see the `echo` command followed by our seven-word sentence—"Our cat Luna is the ranch manager." This means that seven arguments are passed to the `echo` command (with the period as part of the seventh argument). Then, the `>` (greater-than) sign is telling the shell to send the output of the `echo` command into a file named `herding-cats.txt`, or create the file if it does not exist. This is really powerful! The `>` (greater-than) sign is acting as a *redirect operator*, which we will discuss below. If you then open the `herding-cats.txt` file using any common text editor application such as *TextEdit* on a Mac, *Text Editor* on Linux, or *Notepad* on Windows, you'll see the contents of the file, as shown in *Figure 25*.

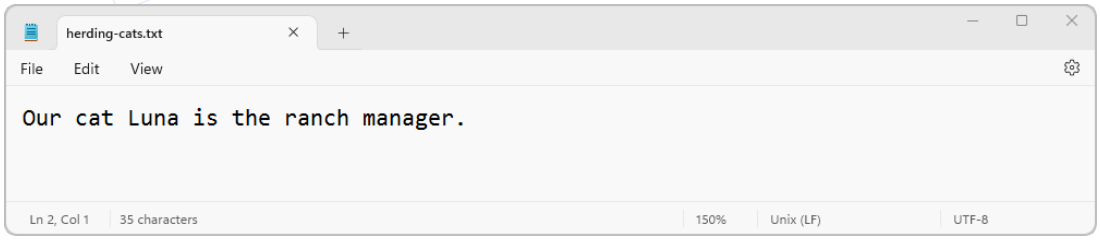

Figure 25. Viewing the herding-cats.txt file using a graphical text editor on Windows.

Redirection, Standard Input, Output, and Error

In Unix-like operating systems, there is the concept of *redirection* that allows us to send the output of a command to other destinations, including into a file. As we can see, this is super helpful! There are a few characters that we use in our shell commands to enable this, and they are known as *redirect operators*. We've already seen that the `>` (greater-than) sign will redirect the command output into a file specified on the right side of the operator, and it will create the file if it doesn't already exist. You may be asking—What if you've already written something to a file, and you want to add to the file instead of overwrite it? We have the magic! A `>>` (double greater-than) sign acts as a redirect operator, but appends to the file instead. Let's give that a try! *Example 29* shows the append operator.

Example 28: Using the `echo` *command to append output to a file on Windows WSL.*

```
chris@DESKTOP-L7H0RFS:/mnt/c/Users/chris$ echo She runs the show! >>
herding-cats.txt
```

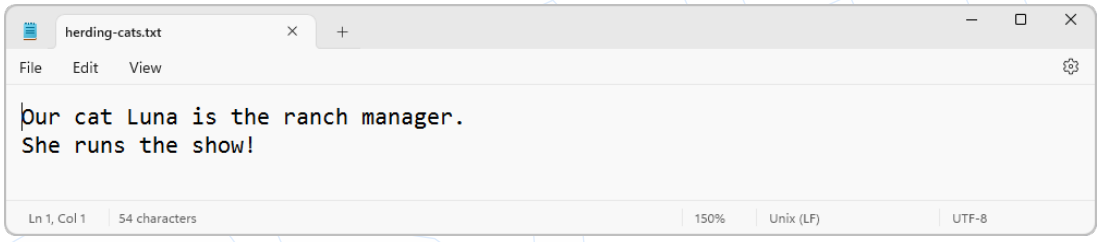

Figure 26. Viewing the appended herding-cats.txt file using a graphical text editor on Windows.

Perfect, we now understand how easily redirection can be used to create, overwrite, or append to files, and that the command syntax is very simple. But what happens when we issue this same command, but without any redirection? Let's try it, this time on a Mac, as shown in *Example 30*.

Example 29: Using the `echo` *command to display text in the terminal window on a Mac.*

```
chris@ophir ~ % echo Our cat Luna is the ranch manager.
Our cat Luna is the ranch manager.
```

Well that is very straight-forward! We're echo-ing whatever we type as arguments to the command in order to display it as output in the terminal window. In this case we are passing those same seven arguments to the command. Let's also run it on Linux, but this time with quotes, seen in *Example 31*.

Example 30: Using the `echo` command to display quoted text in the terminal window on Linux.

```
chris@nuthatch:~$ echo 'Our cat Luna is the ranch manager.'
Our cat Luna is the ranch manager.
```

Hah, interesting! We get the same output. When we issued this second command with quotes, the shell interprets the text as a single *string* argument being passed to the `echo` command.

In computer programming, a sequence of text-based characters is referred to as a *string*. It can include letters, numbers, punctuation, spaces, symbols, etc., and is treated as a single unit.

We see in the previous examples that the `echo` command sends output to the terminal window, which is very useful in scripts when you want to display progress as the script runs.[1] This is the reason why the command is named `echo`, but we are now aware of how powerful it is when combined with redirection.

Looking at the manual pages for the `echo` command on Mac and Linux, they differ slightly. They state "Write arguments to the standard output.", and "Display a line of text.", respectively. Have a look at the manual page to get a feeling for the command. But what is the *standard output*?

When you open a terminal window and the shell interpreter starts up, it creates a few resources that are called *file descriptors*, which help with input and output operations. These include *standard input*, *standard output*, and *standard error*. As you can see, the terminal window itself acts as the device where the shell sends standard output (meaning the default output location). If you make a mistake when typing your command, the messages that get written to standard error also show up in the terminal window by default. We will discuss the idea of standard input in more detail shortly, but think of what you type on your keyboard as an example of standard input for the shell interpreter. Just know that these mechanisms represent the input source, output destination, and error destination when issuing a command.

You will often see these three file descriptors written as *stdin*, *stdout*, and *stderr* respectively.

We can visualize a typical flow of information on the command line as inputs to a *process* and the outputs of that process, shown in *Figure 26*. This diagram illustrates the flow of information as inputs and outputs whenever you issue a command.

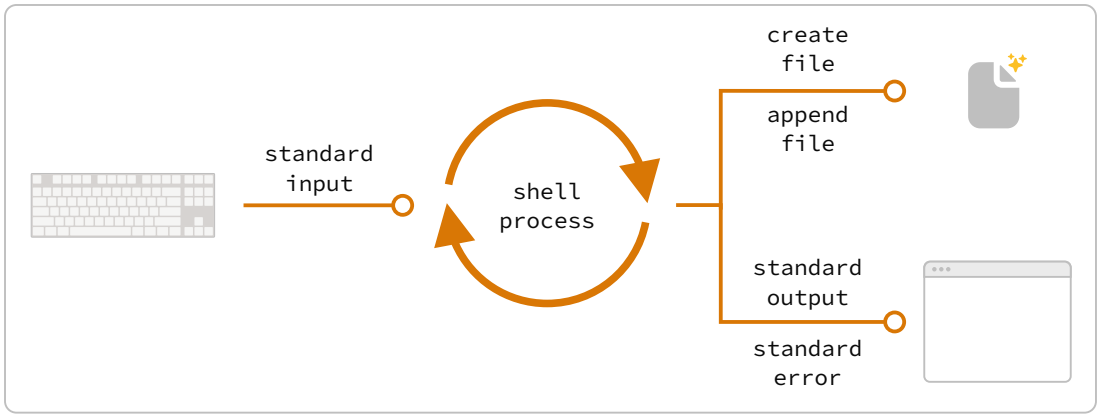

Figure 27. Visualizing inputs and outputs when issuing commands in an interactive terminal shell. Standard input from a keyboard is processed, and is sent by default to the terminal window as standard output and standard error. With redirection, the output can be redirected into a file.

We now have a solid understanding of the `echo` command and how to use redirection on the command line to quickly create or append to a file. We also see that the standard input, standard output, and standard error file descriptors help with input and output operations on the command line, and that the terminal iteself acts as both the standard output and standard error destination. Let's now focus on working with the file that we created on the command line, to highlight how we can effortlessly move the file, copy it,and delete it.

The mv Command—Renaming and Relocating Files

The `mv` command is part of a suite of file handling commands that allows us to move a file from one location to another in the file system. If you use the `mv` command to move the file into the same directory but with a different name, you are effectively just renaming the file. Let's try this with our newly-created `herding-cats.txt` file. *Example 32* demonstrates moving a file.

Example 31: Using the `mv` command to rename a file on Linux.

```
chris@nuthatch:~$ mv herding-cats.txt ranch-manager.txt  ❶
chris@nuthatch:~$ ls -tr ❷
Videos     Music      Downloads        weekly-data.csv
Templates  Desktop    Pictures         daily-data.csv
Public     Documents  yearly-data.csv  ranch-manager.txt
```

❶ Move `herding-cats.txt` to `ranch-manager.txt`.

❷ List the files in reverse chronological order to show the renamed file.

Excellent! That is a quick way to rename a file! Also notice that you can use tab completion when typing the `herding-cats.txt` file name to save you from typing it out completely. Just type `herd` followed by the `Tab` key, and the file name will instantly fill in for you on the command line! Tab completion is super helpful! You then just need to type your destination file name.

We now see a the `ranch-manager.txt` file, and can open it in a text editor application, showing that the contents are the same as the `herding-cats.txt` file, as shown in *Figure 27*.

Figure 28. Viewing the contents of the `ranch-manager.txt` *file on Linux.*

To build on our familiarity of the `mv` command, let's also move multiple files at the same time. In this example, we will first practice with the echo command to create two more files, and then move our three files into the `Desktop` directory. On Windows, be sure to change directories into your Windows home directory where your `Desktop` directory is. *Example 33* shows how to move multiple files.

Example 32: Using the `mv` *command to move multiple files at once on Linux.*

```
chris@nuthatch:~$ echo 'Luna runs a tight ship.' > \
ranch-manager-2.txt ❶
chris@nuthatch:~$ echo 'The horses give Luna plenty of room.' > \
ranch-manager-3.txt  ❷
chris@nuthatch:~$ mv ranch-manager.txt ranch-manager-2.txt \
ranch-manager-3.txt Desktop/  ❸
chris@nuthatch:~$ ls -tr ./Desktop ❹
ranch-manager.txt  ranch-manager-2.txt  ranch-manager-3.txt
```

❶ Create a second file using a multi-line command with the `\` backslash escape. This isn't required.

❷ Create a third file.

❸ Move the three files into the Desktop directory.

❹ List the files in the `Desktop` directory

Using the `mv` command will overwrite any file with the same name in the destination directory, so be careful with this command.

It is important to note that the `mv` command is equally as powerful as the `echo` command coupled with redirection. If you are moving a file to another directory with the same file name, it will overwrite the file, no questions asked! To be more cautious with this command, you can use the `-i` or `--interactive` options, which tells the `mv` command to prompt you for confirmation if it will end up overwriting an existing file. Have a look at the manual page for the details and options for the `mv` command. *Example 34* shows how to move a file with the interactive option.

Example 33: Using the `mv` command interactively to avoid overwriting an existing file on Linux.

```
chris@nuthatch:~$ cd Desktop/
chris@nuthatch:~/Desktop$ mv -i ranch-manager.txt ranch-manager-2.txt
mv: overwrite 'ranch-manager-2.txt'? n  (1)
chris@nuthatch:~/Desktop$
```

(1) Answering `n` or `no` will stop the `mv` command. Answering `y` or `yes` will continue with the command.

Now that we know how to rename and move files, let's turn our attention to copying files, which is also very fast via the command line.

The cp Command—Copying Files

In order to copy a file, we use the `cp` command, and yes, it is as simple as it sounds. We copy one source file name to a destination file name, and *Example 35* shows the simple syntax.

Example 34: Using the `cp` command to copy a file on Linux.

```
chris@nuthatch:~$ cd Desktop/
chris@nuthatch:~/Desktop$ cp ranch-manager.txt ranch-manager-4.txt (1)
chris@nuthatch:~/Desktop$ ls -tr
ranch-manager.txt    ranch-manager-2.txt
ranch-manager-3.txt  ranch-manager-4.txt
```

(1) Copy the source file name to a destination file name

It's really that easy! And like the `mv` command, there is also a `-i` interactive option to insure you are

aware of overwriting any destination files because the `cp` command will otherwise immediately copy the file. Very powerful! In *Example 36*, we copy multiple files to another directory in order to create a backup of the files, and we do this interactively with a multi-line command to keep it tidy.

The `cp` command is just as powerful as the `mv` command, and will overwrite any existing destination file names, so be careful with this command, and use the `-i` option to prompt before overwriting files.

Example 35: Using the `cp` command to interactively copy multiple files on Linux.

```
chris@nuthatch:~/Desktop$ cp -i ranch-manager.txt \
> ranch-manager-2.txt ranch-manager-3.txt \
> ranch-manager-4.txt ~/Downloads  (1)
chris@nuthatch:~/Desktop$ ls -tr ~/Downloads
ranch-manager.txt    ranch-manager-3.txt
ranch-manager-4.txt  ranch-manager-2.txt
```

(1) Since the files didn't exist in the `~/Downloads` directory, we are not prompted about overwrites.

Of course, there may be situations where the destination directory that you are copying to doesn't exist. In this case, the `cp` command will print an error message to standard error, meaning it will show you in the terminal. *Example 37* shows a typical error when the destination directory is missing.

Example 36: Showing the failure of a `cp` command when the destination directory doesn't exist on Linux.

```
chris@nuthatch:~/Desktop$ cp ranch-manager.txt \
> ranch-manager-2.txt \
> ranch-manager-3.txt \
> ranch-manager-4.txt \
> ~/Backups
cp: target '/home/chris/Backups': No such file or directory  (1)
```

(1) The shell lets you know the `Backups` directory doesn't exist

In *Chapter 4. Folder Commands* we will learn how to create a directory via the command line, which will solve our issue shown in *Example 37*, but just know that a destination directory needs to exist when copying multiple files. Now, there is a curious edge-case that may happen when copying a single file to another directory when the directory doesn't exist. Let's demonstrate this in *Example 38*.

Example 37: Using the `cp` command to copy a file to a directory on Linux. The result is a new file.

```
chris@nuthatch:~/Desktop$ cp ranch-manager.txt ~/Backups
chris@nuthatch:~/Desktop$ ls -lh ~/Backups
-rw-rw-r-- 1 chris chris 54 Feb  2 14:52 /home/chris/Backups  (1)
```

(1) A file called `Backups` is created in the home directory

Wait, what happened? When the directory doesn't exist while copying a single file, the shell interprets the `cp` command as a file-to-file copy, rather than a file-to-directory copy! While our intention was to create a backup of the file in the `Backups` directory in our home directory, it instead just created a file called `Backups`. This just shows how the command line will do exactly what you tell it, even when you may have had a different intention!

Speaking of intention, let's be very intentional in the next section on removing files, because the `rm` command is no joke!

The rm Command—Deleting Files

In this chapter we have learned how to create, move, and copy files thus far, and inevitably we will want to quickly and concisely remove files when we make a mistake, or when we just want to create more space for storage. The `rm` command is your friend! And like the `echo`, `mv`, and `cp` commands, it will dutifully delete whatever you tell it to delete, no questions asked. Thankfully, the `rm` command also has the `-i` and `--interactive` options available to you, which certainly generates some peace of mind. Go ahead and have a look at the manual page for the `rm` command to get a sense of the syntax, but it is mighty easy, as we can see in *Example 39*.

The `rm` command is also very powerful, and will immediately delete the files you provide as arguments. There is no concept of a *Trash* can or *Recycle Bin*, so be careful with this command, and make it a habit to use the `-i` option to prompt before deleting files.

Example 38: Removing a file with the `rm` command on Linux.

```
chris@nuthatch:~/Desktop$ cd ~
chris@nuthatch:~$ rm -i Backups
rm: remove regular file 'Backups'? yes
```

Super easy! There are many times that we download huge files from the Internet that we no longer need, or perhaps it was the wrong file afterall. We may have thousands of camera image files, or huge video files that need to be deleted. With a bit of mindfulness, the `rm` command can make your life much easier when it comes to cleaning up unwanted files quickly. In *Chapter 4. Folder Commands*, we will see how we can remove directories as well with the `rmdir` command, but the `-r` option for the `rm` command will do the same. To keep it simple here, we'll address folder removal in the next chapter.

While we have organized the `mv`, `cp`, and `rm` commands in this chapter called *File Commands*, they can also be used with folders, which we will show in *Chapter 4. Folder Commands*.

You are likely seeing a common pattern with each of these commands that we use to manage files—they work equally well on multiple files as they do on a single file. This is no different when deleting files, and *Example 40* just demonstrates how to remove multiple files quickly. Remember from our earlier examples that you can always use tab completion to quickly build a list of files to delete. As a refresher, just type a few of the beginning characters of a file name and then press the `Tab` key to let the shell complete the file name for you. So efficient!

Example 39: Removing a multiple files with the `rm` command on Linux.

```
chris@nuthatch:~$ cd ~/Desktop/
chris@nuthatch:~/Desktop$ rm -i ranch-manager-2.txt ranch-manager-3.txt
ranch-manager-4.txt
rm: remove regular file 'ranch-manager-2.txt'? yes
rm: remove regular file 'ranch-manager-3.txt'? yes
rm: remove regular file 'ranch-manager-4.txt'? yes
chris@nuthatch:~/Desktop$ ls -tr
ranch-manager.txt
```

Notice the interactive option confirms the removal of each file individually. This works for a few files, but is untenable for hundreds or thousands of files. In these cases, double check your command is correct, and forego the `-i` interactive option, and your files will be deleted instantly. I'm sure you're wondering— *Do I have to type out my thousands of file names in order to delete them?* Definitely not! In *Chapter 4. Folder Commands*, we will explore the concept of *expansion* on the command line, where we can use wildcard characters and other tricks that allow us to use a pattern (for example, all files ending in `.jpg`) to create a list to delete, copy, or move. For now, let's round out our file handling commands and learn how to edit files directly in a terminal window using the `nano` command.

The nano Command—Creating and Editing Files

At the beginning of this chapter we familiarized ourselves with the `echo` command and used redirection operators to create, overwrite, and append to a file. This is great for quick file work, or for appending progress lines to a file, and similar lightweight work. However, it's nice to have more flexibility while writing, particularly if you are writing a document or editing a configuration file on a remote server. This is where common editor commands like `nano` shine! While most of the time it is easiest to use a graphical text editor such as TextEdit on a Mac, Text Editor on Linux, or Notepad on Windows, there are sitations where a terminal-based editor like `nano` is a very helpful tool. Let's explore the `nano` command first by editing our `ranch-manager.txt` file that remains in our `Desktop` directory. *Example 41* shows how to open a file with the `nano` command.

While the `nano` command is available on Mac, Linux, and Windows WSL, the Mac version currently is an alias to the `pico` command. Typing `nano` on a Mac will therefore open the `pico` editor.

Example 40: Editing a file with the `nano` *command on Linux.*

```
chris@nuthatch:~$ cd Desktop
chris@nuthatch:~/Desktop$ nano ranch-manager.txt
```

By just passing the file name as an argument to the `nano` command and pressing the `Return` key, we are presented with an editor view that fills the terminal window. Give this a try! We will walk through the steps of how to edit a file with `nano`. *Figure 28* shows an example of editing a file.

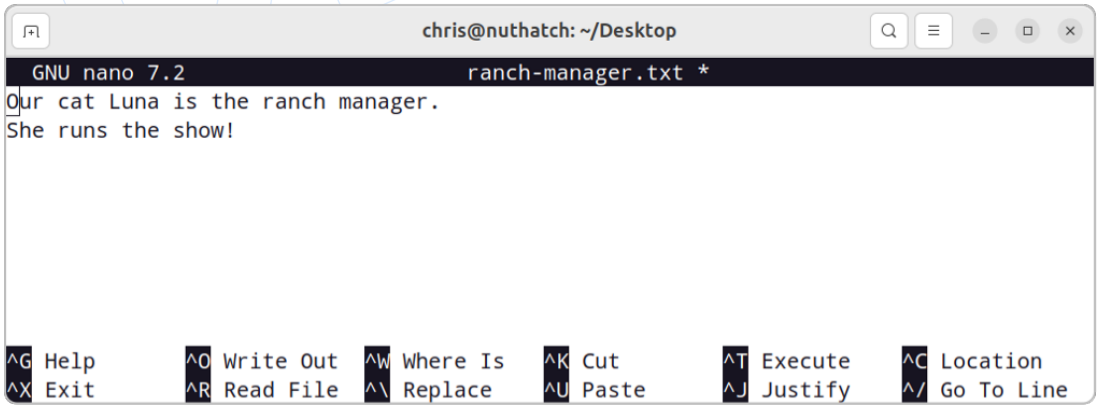

Figure 29. Opening the `ranch-manager.txt` *file using the* `nano` *command.*

You'll see that our terminal window turns into a small editor with a dark header section at the top, and a menu of key combinations in the footer section at the bottom. The blank space in the middle is where you write. We are placed on the first line of the file, and we can just start typing! Editing is that easy, and moving the cursor is intuitive with the four arrow keys on your keyboard—`◄` (left arrow), the `►` (right arrow), the `▲` (up arrow), and the `▼` (down arrow). After adding a couple of lines to the text file, let's save the file. To do so, let's familiarize ourselves with the menu at the bottom of the editor. Depending on how wide your terminal window is, you will see two rows of menu item shortcuts with a key combination next to the label. The `^` (caret) symbol is shorthand for the `Control` key, and is used in combination with another letter, number, or symbol to do things like save the file, search in the file, cut, paste, etc. So in our case, we'll press `^O` (`Control`+`O`) to "Write Out" the file (meaning save it to disk). You can use lowercase or uppercase letters with these key combinations. *Figure 29* shows this menu item in progress. Try it yourself!

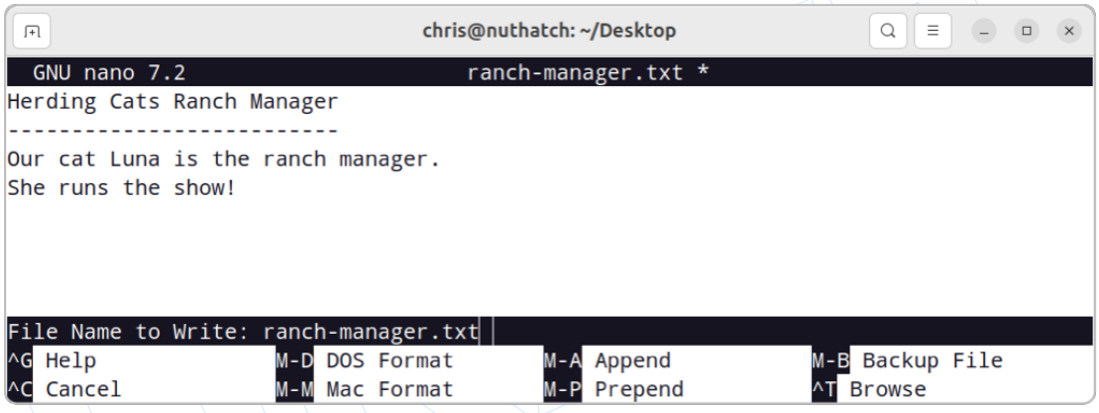

Figure 30. Saving a file using the `nano` `^O` *menu item (*`Control`+`O`*) key combination.*

The editor will prompt you to confirm the file name you want to save to, defaulting to the current file name. Just pressing the `Return` key will save the file. It's that easy! We've now edited our first file via the command line!

To orient ourselves more with the menu for the `nano` and `pico` commands, notice that some menu items begin with an `M-` (M and dash characters), followed by another character. This is shorthand for the `Meta` key on your keyboard, which on modern computers is mapped to the `Esc` (escape) key. In the `nano` editor, you double press the `Esc` key (Escape key twice) followed by the letter for the menu item. That's a bit obscure, but once you understand it, it is easy to use. Let's clarify this with an example using the *Undo* menu item, which is labeled with a `M-U`. Go ahead and delete a line of text using the `^K` (`Control`+`K`) key combination. Then, press the `Esc` (Escape) key twice, followed by the letter `K`. The line you deleted should show back up in the file because of this "undo" action.

If your terminal window is not very wide, some of the menu item key combinations in the lower window will be hidden, as is the case in *Figure 29*. You can always use the `^G` (`Control`+ `G`) combination to show the help menu for the `nano` command. The help menu also uses the keyboard arrow keys to scroll. When you are finished using the editor, type `^X` (`Control`+ `X`) to exit the `nano` editor and get back to your command prompt.

Redirecting a here-document

Now that you are familiar with editing a file using `nano`, let's explore one more way of using the `nano` command that builds on our knowledge of using redirection operators from earlier in the chapter. In this case, let's say we want to begin writing a document on the command line, and redirect the text into the `nano` command for further editing. This is a fun edge case, and it illustrates a command line concept called a *here-document*. *Example 42* shows how to create a *here-document* and redirect it into the `nano` editor for further editing.

Example 41: Redirecting a here-document on the command line into the `nano` *editor on Linux.*

```
chris@nuthatch:~$ nano - << EOF
> Our cat Luna woke us up at 5 am.
> She's an early riser.
> Thanks Luna. :)
> EOF
```

Wow, that's a curious one! Let's break this down. We first have the `nano` command with a single `-` (dash) argument. This is a `nano` option to read from the standard input, meaning anything that is typed on the keyboard. The `<<` (double less-than) operator is redirecting what is called a *here-document* (anything typed on the command line) until it encounters an `EOF` token (meaning end-of-file). The two `EOF` tokens act as bookends to our here-document, and once we type the second `EOF` followed by the `Return` key, everything we type will be passed into the `nano` editor. Now that is some magic!

Similar to a multi-line command, the shell places `>` (greater-than) signs on the left of the terminal to indicate you are still typing a here-document. Note that `zsh` on a Mac will add a `heredoc>` indicator.

The result of our finished command is shown in *Figure 30*, where we can continue to edit the document in the `nano` editor.

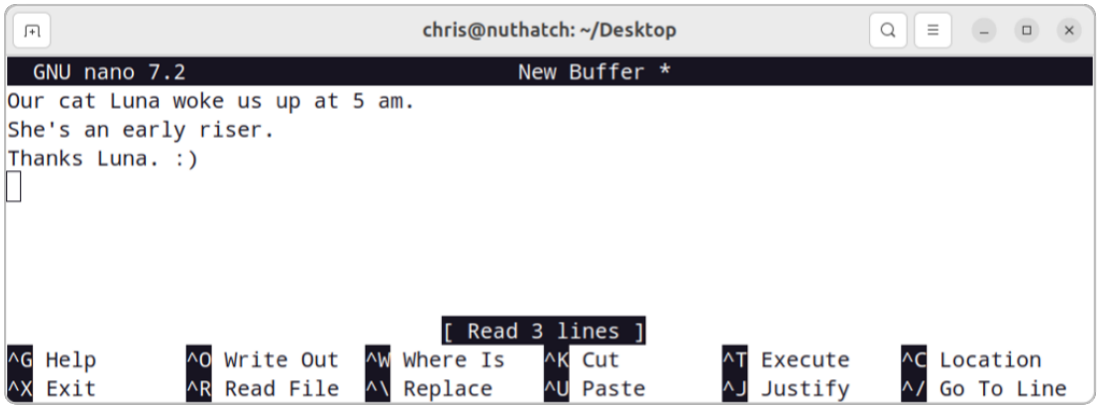

Figure 31. Continuing to edit a here-document passed into the `nano` *command editor.*

Notice that the `nano` command read the three lines from *stdin*, and created a new in-memory buffer for our file, but the file is not saved to disk yet. We can edit the file and save it normally from here. Also note that the `EOF` tokens that act as beginning and ending delimiters to our here-document can be any single word, and `EOF` is just used as a convention. If we wanted, we could use `LunaIsHungry` twice as a token to bookend our document. Feed the cat!

This wraps up our mini-journey with the `nano` command and editing via the command line. It's also good to be aware that there are other very popular command line editors, such as `vi` (or `vim`) and `emacs`. Both are very powerful, so feel free to explore them as you build your command line skills. Onward!

Command Line File Handling is Awesome!

We have really made awesome progress in our command line journey by exploring the file handling commands that build on top of the core commands we learned in the last chapter. We are now familiar with easily creating files with the `echo` command via the handy command line concept of redirection. We can magically append to a file, or overwrite a file, depending on our needs. So powerful! We also understand that the special file descriptors called standard input, standard output, and standard error are available to us as the default locations of data flow on the command line (from the keyboard to the terminal, or optionally redirected). This is amazing stuff! As we have files to work with, we are now fully acquainted with copying (`cp`), moving (`mv`), and deleting (`rm`) files—everyday tasks that can be accomplished extremely quickly from the command line. Likewise, we have a solid understanding of how to edit a text file directly within the terminal application using the `nano` command. We now understand the concept of a here-document, and can redirect text into a command that can read from standard input. Seriously clever! All of these file handling skills allow us to manage files on servers around the world, or efficiently work with our own files locally. These commands work in concert with our core commands discussed earlier, and with the folder-oriented commands in *Chapter 4. Folder Commands*. See you there!

[1] Commands like those we are learning about can be written into shell scripts, which are files that can be run like programs from the command line. They are often named similar to `myscript.sh` where the `.sh` file ending indicates that the file uses the shell programming language.

Folder Commands

mkdir, rmdir, du

Lightning Fast Folder Management

Everything we create on a computer involves a folder in one way or another. It is the quintessential means of organizing everything we store digitally. In the previous chapters we have changed into directories using the `cd` command, listed directory contents using the `ls` command, and created files inside of directories with the `echo` and `nano` commands. We've made incredible progress! In this chapter, we will build our foundation even further by exploring the commands needed to manage directories. In particular, we will use the `mkdir` command to make new directories, and the `rmdir` command to remove them. We'll also explore the `du` command, which gives us a powerful tool to understand our disk usage, and where all of those huge files are that are taking up storage space on our computers. Given our previous experience with managing files, we will see that these directory-level commands are just as easy to use, and are just as powerful! In fact, we'll look at an interesting command line concept called *expansion*, which allows us to use special bracketing characters to generate directory or filenames, and wildcard characters to match file and directory name patterns. These techniques make managing files and directories lightning fast, which is part of the magic of the command line. We'll also revisit the `cp`, `mv`, and `rm` commands that we learned in *Chapter 3. File Commands*, because they are equally useful for directories! Let's get organized!

The mkdir Command—Creating Directories

Let's jump right in because it doesn't get any easier to create a directory than with the `mkdir` command. The synopsis in the manual page says it all— "make directories"! *Example 43* shows the simplest use of the `mkdir` command.

Example 42: Making a directory with the `mkdir` *command on macOS.*

```
chris@ophir ~ % mkdir Photography
chris@ophir ~ % ls -tr
Music       Library     Movies      data.txt    Desktop
Public      Documents   Downloads   Pictures    Photography
```

That's it! We've made our first directory called `Photography`, and because our current directory is our home directory, we see the `Photography` directory gets created right where we specified using a relative directory path. But let's think a bit about how we need to get organized. Let's say we have many hundreds of photos on our camera that we just imported into a single directory, and we want to categorize them by location and by date so we can work on them as daily shoots. If we know the locations and the dates, we could begin to create our directory structure by adding the `-p` option to the `mkdir` command, which causes it to create intermediate subdirectories if they do not exist. *Example 44* demonstrates making a hierarchy of directories using a single command.

Example 43: Making a hierarchy of directories with the `mkdir` *command on macOS.*

```
chris@ophir ~ % mkdir -p Photography/Shoots/Utah/2024/07/04
```

With a one-line command, we have now created five more directories inside our `Photography` directory. So easy! *Figure 32* shows the result using a Finder window on macOS.

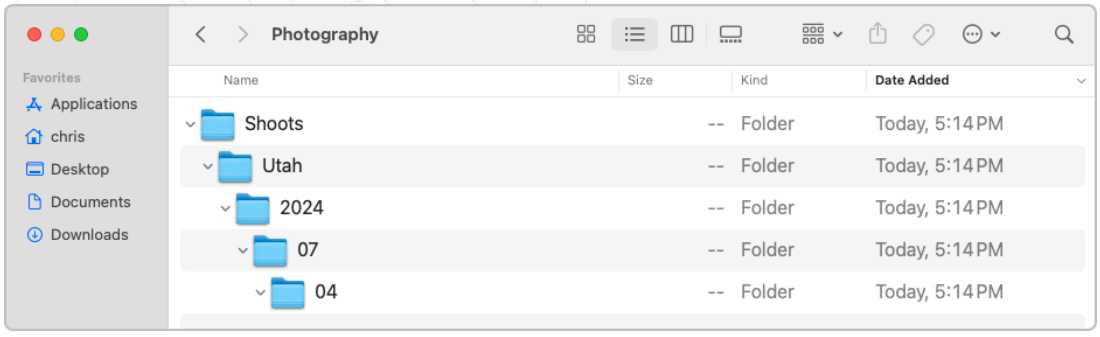

Figure 32. Showing the results of creating a directory hierarchy with the `mkdir` *command on macOS.*

This is a great example of how fast a command can be in accomplishing a lot of work. However, what if you have photography shoots at dozens of locations and dates, and need to create all of the yearly, monthly, and daily sub-folders for those sessions? We have the magic! In the next section, let's explore the concept of command line expansion, and how it will help us get the job done quickly.

Expansion—Powerful Techniques To Speed Up Your Commands

The terminal shell gives us a number of shorthand techniques to generate file and directory paths, and they generally fall under the topic of *expansion*. To understand expansion, let's first review a type of expansion that we have already covered— *tilde expansion*. We understand that a `~` (tilde) sign on the command line signifies "the user's home directory". So if we type `cd ~`, the command expands to be `cd /Users/<your-username>` on macOS or `/home/<your-username>` on Linux and WSL. Likewise, using `~venus` on the command line means "the username *venus*' home directory", which expands to `/Users/venus` or `/home/venus`, depending on the operating system. So expansion generally gives us shorthand syntax that expands to longer file or directory paths. So let's first explore the concept of *brace expansion* to help us with our photo organization project from above.

Brace expansion

Brace expansion allows us to define either a list or a sequence in our file or directory paths, and we'll cover both scenarios here. Let's first explore brace expansion with a list. The word *brace* refers to the `{}` (curly brace) symbols that are used to bracket phrases or other content. In our photography project, we have photos from multiple locations that we need to organize, and on multiple dates. Let's first create the locations in the directory structure using brace expansion. *Example 45* demonstrates brace expansion by creating directories for four US states where the photo shoots took place in our existing `Photography` directory.

Example 44: Using a list brace expansion with the `mkdir` command on macOS.

```
chris@ophir ~ % mkdir -p Photography/Shoots/{Arizona,Colorado,New-
Mexico,Utah}
```

Wow! In a single command, we just created four directories within the `Shoots` subdirectory of our `Photography` directory. Let's break this down. Using the curly braces as the cue to the shell that we are going to use expansion, we provide a comma-separated list of directory names that we want to create. These are expanded in the shelll to represent the following four paths:

- `Photography/Shoots/Arizona`
- `Photography/Shoots/Colorado`
- `Photography/Shoots/New-Mexico`
- `Photography/Shoots/Utah`

Each path is passed as an argument to the `mkdir` command to create these directories. And notice that while we are asking it to create the `Utah` folder in this command, the `-p` option will create directories that don't exist already, so it is skipped. Very convenient! *Figure 33* shows the results of our command.

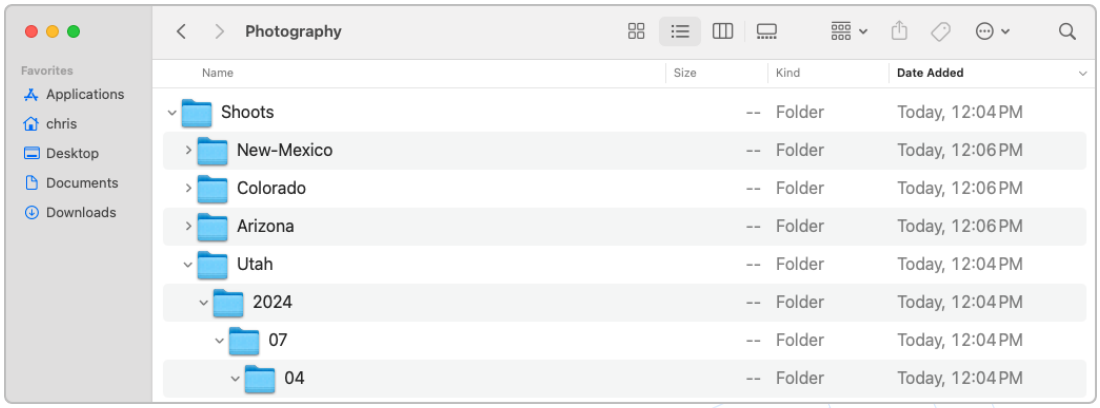

Figure 33. Showing the results of creating a directory hierarchy with brace expansion on macOS.

Let's now continue with building our directory structure by generating folders for the dates of the photo shoots. In *Example 44* we created a directory structure for the photo shoot on July 4, 2024 in Utah, represented by the `Photography/Shoots/Utah/2024/07/04` directory path. Assuming we have had photo shoots over the last five years in our four locations on different days of every month of the year, let's use expansion to generate the directories systematically. *Example 46* demonstrates how to use a *sequence expression* within a brace expansion to create years, months, and days automatically.

Example 45: Using list brace expansion and sequence expansion with the `mkdir` *command on macOS.*

```
chris@ophir ~ % mkdir -p Photography/Shoots/{Arizona,Colorado,New-
Mexico,Utah}/{2020..2025}/{01..12}/{01..31}
```

Whoa! What just happened? We just generated 9,246 directories that represent each day of each month of the years 2020 to 2025 for each of our photo shoots in the four corner states in the US. Incredible! Go ahead and browse the directory structure on your machine to see the results, and let's discuss the sequence expression syntax.

We first see the `{2020..2025}` part of the path. This cues the shell to create a list of sequential numbers from `2020` to `2025` and generate sub-directories for each year. The same syntax is used for the monthly directories, where the `{01..12}` sequence tells the shell to create a list of numbers from `01` to `12` representing the months of the year. Notice that by adding the zero before the number one (`01`), the shell knows to zero-pad all of the single digit months so that all sub-directories are created

with two digits. And lastly, the `{01..31}` portion of the path tells the shell to generate a list of daily folders from `01` to `31` for each daily directory. Now that is some lightning fast folder creation!

We can now organize our photos into the appropriate directories for the given days and locations that they occured. While obviously it would be impossible to have a photo shoot of every day of every month for five years, this is an exercise in demonstrating how easy it is to generate a directory structure that suits your needs with just a couple of expansion tools in your toolbox. You probably also noticed that we generated too many days for many of the months that only have 28, 29, or 30 days in them. We'll have a look at efficiently removing directories in the section on the `rmdir` command, but let's first learn to create a backup of our work in case any problems arise.

Copying and renaming directories

In our photo shoot scenario, let's say we've added JPEG photo files into the directories that coincide with the dates of our photography shoots. These include the following locations and dates:

Table 3. Our ficticious photography shoots from 2020 to 2025 in the four corners states.

Year	Arizona	Colorado	New-Mexico	Utah
2020	Jan 01, 02	Feb 10, 11	Mar 16, 17	Apr 22, 23
2021	May 30, 31	Jun 05, 06	Jul 13, 14	Aug 24, 25
2022	Sep 29, 30	Oct 05, 06	Nov 14, 15	Dec 21, 22
2023	Jan 03, 04	Feb 09, 10	Mar 17, 18	Apr 24, 25
2024	May 01, 02	Jun 08, 09	Jul 16, 17	Aug 20, 21
2025	Sep 27, 28	Oct 05, 06	Nov 13, 14	Dec 22, 23

In our scenario, the shoots for 2025 are scheduled but not complete, and so we've created a file called `reserved.txt` in each of those date directories as placeholders.

It is a great practice to back up your work before issuing commands that could have big consequences. As mentioned in *Chapter 3. File Commands*, the `cp` command can be used for directories because it has a `-R` recursive option. Let's create a backup of our `Photography` directory for safe keeping. *Example 47* demonstrates this.

Example 46: Using the `cp` command recursively to backup a directory on macOS.

```
chris@ophir ~ % cp -R Photography Photography-Backup
chris@ophir ~ % ls -tr
```

```
Music           Documents       data.txt        Photography
Public          Movies          Pictures        Photography-Backup
Library         Downloads       Desktop
```

Excellent! We now have a pristine copy of our `Photography` directory saved to the side to preserve our work.

The rmdir Command—Deleting Directories

Removing directories is as easy as creating them, and as we've mentioned, the `rmdir` command is our tool of choice. Go ahead and look at the manual page for the `rmdir` command to get to know the command options. The synopsis states— "remove empty directories". The convenient aspect of this command is that it cautiously removes directories, avoiding any directories that have files inside of them. That can be invaluable!

If we tried to issue a command like `rmdir Photography`, based on our knowledge of the command, it wouldn't remove anything because it has sub-directories inside of it, some of which have photo files based on the dates in *Table 3*. So we want to pass a list of all of the sub-directories as arguments to the `rmdir` command, and use the `-p` option to evaluate each sub-directory separately. *Example 48* uses the same brace expansion in our `mkdir` commands previously to generate a list.

While the `rmdir` command only removes empty directories, it still permanently removes them. Be sure of the directories you want to remove.

Example 47: Using list brace expansion and sequence expansion with the `rmdir` command on macOS.

```
chris@ophir ~ % rmdir -v -p Photography/Shoots/{Arizona,Colorado,New-
Mexico,Utah}/{2020..2025}/{01..12}/{01..31}
rmdir: Photography/Shoots/Arizona/2020/01/01: Directory not empty
rmdir: Photography/Shoots/Arizona/2020/01/02: Directory not empty
Photography/Shoots/Arizona/2020/01/03
rmdir: Photography/Shoots/Arizona/2020/01: Directory not empty
Photography/Shoots/Arizona/2020/01/04
...
```

So magical! Let's review what we did here. First, we used the `-v` verbose option so we see which directories were deleted and which were passed over. The `-p` option tells the `rmdir` command to evaluate every directory in the given list. Since the directory path we passed to the command included

brace expansion for the state location directories, and sequence expansion for the month and day directories, the `rmdir` command evaluated every directory we automatically created earlier with the `mkdir` command. In the end, of the 9,246 directories we generated, it left 102 of them (those with files in them). Even the directories with invalid days in the months were removed because they had no files in them. The output in *Example 48* is truncated for display purposes, but you can see how easy it is to remove empty directories in a very complex directory structure without much work at all. Thank you `rmdir` authors!

Know your power

While the `rmdir` command is great for mindful removal of undesired empty directories, there are times when you just need to remove an entire directory completely, regardless of what is inside of it. Of course, we have the power! As mentioned in *Chapter 3. File Commands*, the `rm` command can be used to remove directories. We are able to do this by using the `-R` option, which tells the command to descend into the directory and delete all files and folders. Let's say that our photo shoot scheduled for November of 2025 in New Mexico was canceled, and we need to delete that directory wholesale. *Example 49* demonstrates a recursive removal of a directory with the `rm` command.

The `rm -R` command will instantly delete everything within the directory that is passed in as an argument, so use extreme caution with this command. Double check that the path is correct, and be aware of mistakes with absolute versus relative paths, or with extra or missing characters.

Example 48: Recursively removing a directory and all its contents using the `rm` command on macOS.

```
chris@ophir ~ % rm -R Photography/Shoots/New-Mexico/2025
```

And just like that, the `2025` directory inside of the `New-Mexico` directory is permanently removed with all of its contents. *Figure 34* shows the results of this command.

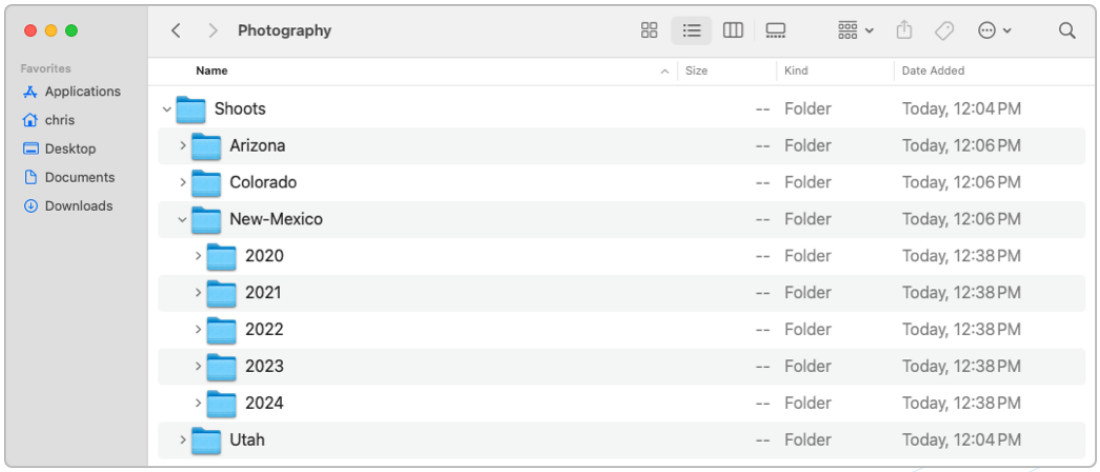

Figure 34. Results of the removal of a sub-directory using the `rm` command on macOS.

The `rm -R` command that we see above can be unforgiving, so be sure use it judiciously! You may unintentionally have a single character typo in your command which might drastically alter the outcome. For instance, if we mistakenly introduced a single space character into our command, the result is quite devastating, as shown in *Example 50*.

Example 49: Beware of typos! Inadvertantly removing a directory with a space character typo on macOS.

```
chris@ophir ~ % rm -R Photography/ Shoots/New-Mexico/2025
rm: Shoots/New-Mexico/2025/: No such file or directory
```

The above command completed, and promptly removed the entire `Photography` directory, and then gave us an error that it could not find a directory named `Shoots/New-Mexico/2025/`! That single space wreaked havoc on our intended command. The take-home message is of course to double check your work when using this command.

Restoring from a backup

In the event that we did make the single space typo mistake shown in *Example 50*, we can then restore our work by moving our backup directory back to our working `Photography` directory, as shown in *Example 51*.

Example 50: Moving (renaming) a directory on macOS.

```
chris@ophir ~ % mv Photography-Backup Photography
```

There we go! We can get back to our work. In the next section, let's explore the ways we can more precisely identify files and folders, which may allow us to skip the recursive option of the `rm` command altogether.

Pathname expansion

We see how powerful the recursive use of the `rm` command can be, and with mindfullness it is our trusty friend. Instead of removing directories entirely, we also have the ability to specify patterns of files and folders to remove with a handy concept called *pathname expansion*. Like brace expansion, the shell will generate a list for us, but in this case, we use special wildcard caharaters in our command. These include the `*` (asterisk, or star), the `[]` (square bracket), and the `?` (question mark) characters, and are affectionately known as *glob operators*. Let's discuss these individually.

Let's say that while we were copying our photo files into our directory structure, we mistakenly copied photos taken in Utah from 2020 to 2023 into our `Colorado` directories, and that we now need to fix this and re-copy the correct photos into place. We know our directory dates are correct, so we just need to selectively remove the JPEG files that are currently there. Let's first use an `*` (star) to specify a file pattern to remove, shown in *Example 52*.

Using wildcard patterns with the `rm` command will permanently delete the files matching the pattern. Double check your work, and list the files with the same pattern before removing them.

Example 51: Using a `*` (star) wildcard to specify a file name pattern on macOS.

```
chris@ophir ~ % ls -tr Photography/Shoots/Colorado/2020/02/10/*jpg
IMG_338.jpg IMG_340.jpg IMG_335.jpg IMG_336.jpg IMG_333.jpg
IMG_339.jpg IMG_334.jpg IMG_337.jpg IMG_332.jpg IMG_331.jpg
chris@ophir ~ % rm Photography/Shoots/Colorado/2020/02/10/*jpg
```

Wow! Let's figure out what happended here. First, we see that there were ten JPEG images in the `Photography/Shoots/Colorado/2020/02/10/` directory. We then used the `rm` command with the `*jpg` (star wildcard followed by the `jpg` file name ending). This wildcard pattern means "any file name string that ends in jpg". That is great, and we instantly deleted those ten files. But we know we have incorrect JPEG images in all of the 2020 to 2023 Colorado sub-directories, so let's explore how we can use square brackets along with the star wildcard to specify the set of files we want to delete.

The `[]` bracket syntax lets us specify a list of single characters to match, but if it includes a `-` (hyphen), it means a range of characters, and the shell will expand them accordingly. *Example 53* uses both the star wildcard and a range of numbers to specify the file paths we would like to remove.

Example 52: Using a `[]` character range and `*` star wildcards to specify a file name pattern on macOS.

```
chris@ophir ~ % ls Photography/Shoots/Colorado/202[0-3]/*/*/*jpg ❶
Photography/Shoots/Colorado/2020/02/11/IMG_321.jpg
Photography/Shoots/Colorado/2020/02/11/IMG_322.jpg
...
Photography/Shoots/Colorado/2021/06/05/IMG_301.jpg
Photography/Shoots/Colorado/2021/06/05/IMG_302.jpg
...
Photography/Shoots/Colorado/2022/10/05/IMG_228.jpg
Photography/Shoots/Colorado/2022/10/05/IMG_229.jpg
...
Photography/Shoots/Colorado/2023/02/10/IMG_288.jpg
Photography/Shoots/Colorado/2023/02/10/IMG_289.jpg
...
chris@ophir ~ % rm Photography/Shoots/Colorado/202[0-3]/*/*/*jpg ❷
```

❶ Listing a set of files using a bracketed range and star wildcards

❷ Removing a set of files using a bracketed range and star wildcards

Now that is some precise file name matching! Let's walk through this command to understand what is happening. *Figure 35* illustrates the patterns matched in the command.

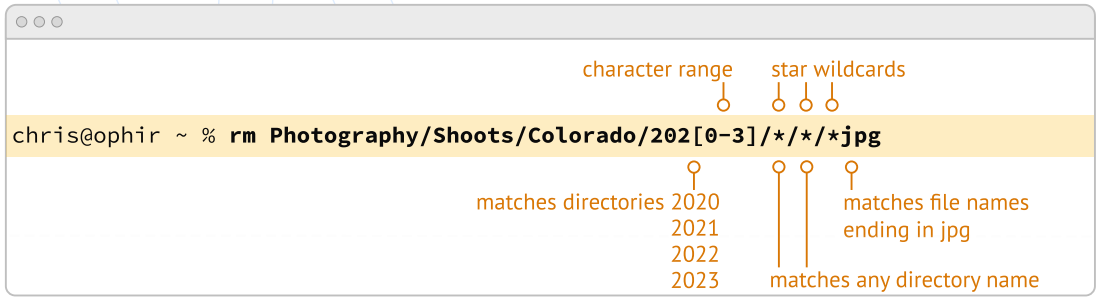

Figure 35. Using pattern matching to enable path expansion in commands on macOS.

In *Example 53* we first use the `ls` command to list the files we want to match as a best practice before doing the removal. The listing is truncated for display purposes, but it has matched eighty files across eight directories. The bracketed range of `[0-3]` expands to the numbers `0, 1, 2, 3`, so the pattern

matches directories named `2020, 2021, 2022, 2023`. This pattern is followed by `/*` star wildcard which will match any directory name (the month directories). This pattern is followed by another `/*` star wildcard which also matches any directory name (the daily directories). Lastly, the `*jpg` star wildcard will match any file names ending in `jpg`. That's it! In one powerful command, we have removed the incorrect images from Utah that were placed into our Colorado directories, and we can now replace them with the correct images.

To round out our pathname expansion toolkit, let's discuss the `?` wildcard. This character matches any single character. So for instance, in the case of our JPEG photo names, a pattern of `IMG_???.jpg` would match any of our image files given how they were named (for example, `IMG_411.jpg`).

There are many other ways to use the various expansion techniques on the command line to make quick work. They are documented in the `bash` and `zsh` manual pages (i.e `man bash`). Just search for the `EXPANSION` section, and in particular the `Pathname Expansion` sub-section to learn about these little gems! But next, let's learn how to get a good grasp on how much disk space we are using in our subdirectories so we can easily manage old, forgotten, and often huge files that are no longer needed. The `du` command is a real eye opener!

The du Command—Viewing Disk Usage

We all have many different files we have created or downloaded and that are no longer needed. When we look at the amount of space left on our storage drives, it seems like we can free up a lot of space if we knew where the unwanted files were located in our file system. The `du` command lets us display our "disk usage" in a number of ways so we can discover where those huge files are located, or just get a sense of where a lot of smaller files are adding up. Let's have a look at the `du` command in the context of our Photography directory. Without a directory argument, the `du` command will calculate usage in your current working directory. *Example 54* demonstrates the command with only the `-h` option to tell it to print human-readable file and directory sizes.

Example 53: Using the `du` *command to view disk usage on macOS.*

```
chris@ophir ~ % cd Photography
chris@ophir Photography % du -h
401M    ./Shoots/New-Mexico/2023/03/18
3.5G    ./Shoots/New-Mexico/2023/03/17
3.9G    ./Shoots/New-Mexico/2023/03
3.9G    ./Shoots/New-Mexico/2023
...
  0B    ./Shoots/Colorado/2025
```

```
484M    ./Shoots/Colorado/2024/06/09
454M    ./Shoots/Colorado/2024/06/08
938M    ./Shoots/Colorado/2024/06
...
458M    ./Shoots/Utah/2022/12/21
549M    ./Shoots/Utah/2022/12/22
1.0G    ./Shoots/Utah/2022/12
1.0G    ./Shoots/Utah/2022
...
588M    ./Shoots/Arizona/2021/05/31
478M    ./Shoots/Arizona/2021/05/30
1.0G    ./Shoots/Arizona/2021/05
1.0G    ./Shoots/Arizona/2021
...
 19G    ./Shoots
 19G    .
```

The listing in *Example 54* has been truncated for display purposes, but we can see that the `du` command has traversed the entire `Photography` directory tree, and has calculated a total size of each sub-directory. It's really fast! We get a look at how big the directories are, and there is a grand total at the end for the `.` (dot) current directory. We see that there are zero bytes in many of the Colorado directories because we recently deleted all of those files in years 2020 to 2023. This command is perfect for directory trees that are moderately sized, but when you have deeply nested directories, you might want to look at a shorter summary to see where the biggest files are. We have the magic! *Example 55* shows the `du` command with the `-s` option which will summarize the directories given as arguments.

Example 54: Using the `du` *command to view summary disk usage on macOS.*

```
chris@ophir Photography % du -sh ./Shoots/*
5.1G    ./Shoots/Arizona
938M    ./Shoots/Colorado
8.3G    ./Shoots/New-Mexico
4.8G    ./Shoots/Utah
```

Ah, so simple! We used a `*` star wildcard in our relative directory path such that each sub-directory (the four corners states) would each be summarized. We see that the `./Shoots/Colorado` directory is much smaller due to our previous deletions, and that the `./Shoots/New-Mexico` directory is the largest by quite a bit. Let's investigate the daily directories in the `New-Mexico` directory to see where the biggest files are located. *Example 56* demonstrates this.

Example 55: Using the `du` command to view summary disk usage in leaf directories on macOS.

```
chris@ophir Photography % du -sh ./Shoots/New-Mexico/*/*/*
533M    ./Shoots/New-Mexico/2020/03/16
529M    ./Shoots/New-Mexico/2020/03/17
587M    ./Shoots/New-Mexico/2021/07/13
730M    ./Shoots/New-Mexico/2021/07/14
521M    ./Shoots/New-Mexico/2022/11/14
497M    ./Shoots/New-Mexico/2022/11/15
3.5G    ./Shoots/New-Mexico/2023/03/17  (1)
401M    ./Shoots/New-Mexico/2023/03/18
522M    ./Shoots/New-Mexico/2024/07/16
672M    ./Shoots/New-Mexico/2024/07/17
```

(1) The largest sub-directory is very apparent

Interesting! So the `./Shoots/New-Mexico/2023/03/17` is definitely an outlier compared to the rest of the sub-directories, so let's view the contents and see the files, as shown in *Example 57*.

Example 56: Viewing a long listing of files in reverse size order on macOS.

```
chris@ophir Photography % ls -lSrh ./Shoots/New-Mexico/2023/03/17
total 7268952
-rw-r--r--  1 chris  staff   2.0M Feb  6 14:00 IMG_180.jpg
-rw-r--r--  1 chris  staff   7.0M Feb  6 14:00 IMG_179.jpg
-rw-r--r--  1 chris  staff   7.0M Feb  6 14:00 IMG_172.jpg
-rw-r--r--  1 chris  staff   9.0M Feb  6 14:00 IMG_174.jpg
-rw-r--r--  1 chris  staff    16M Feb  6 14:00 IMG_175.jpg
-rw-r--r--  1 chris  staff    44M Feb  6 14:00 IMG_177.jpg
-rw-r--r--  1 chris  staff    59M Feb  6 14:00 IMG_178.jpg
-rw-r--r--  1 chris  staff    73M Feb  6 14:00 IMG_171.jpg
-rw-r--r--  1 chris  staff    90M Feb  6 14:00 IMG_176.jpg
-rw-r--r--  1 chris  staff   100M Feb  6 14:00 IMG_173.jpg
-rw-r--r--@ 1 chris  staff   3.1G Feb  7 11:12 DaVinci_Resolve.zip  (1)
```

(1) An errant file downloaded into a deeply nested directory

Ah hah! It looks like there is a mistakenly downloaded Davinci Resolve application installer deep inside the `Photography` directory. We can use the `rm` command to remove that file, as we learned about in *Chapter 3. File Commands*. And with that, we've completed our short tour of the `du` command, which can be very helpful in discovering what exactly is buried in your file system. Feel free

to read the manual page for the `du` command since there are other useful options to help you with managing your directories.

Command Line Folder Handling is Awesome!

While this chapter features only a few commands, they are a total power house when it comes to managing files and directories at lightning speeds. We are now familiar with the ultra-handy `mkdir` command, which allows us to generate entire folder structures with a very compact syntax. When we couple it with the concept of brace expansion, we can automatically generate directory trees with ease. We are also now acquainted with its sister command of `rmdir`, which helps us keep our directories neat and free of unneccessary empty directories. The combination of these two commands allows us to organize our files in ways that are not possible using graphical tools, particularly when using our pathname wildcard expansion techniques. In addition, we now know our power when it comes to wholesale deletion of directories using the recursive option of the `rm` command. As an insurance policy when using the recursive `rm` command, we now have solid strategies to backup and restore our work using the `cp` and `mv` commands. The more we use our systems, the more chances there are for files that are unneeded or obsolete over time. To that end, we've learned that the `du` command is a powerful way to understand what is taking up our disk space, and we now have the power to trim things down! What a whirlwind! In the next chapter we will continue to discover how command line file management is at our fingertips, with a focus on data-oriented commands that let us delve inside of files. We're heading to the stars!

Text Data Commands

cat, sort, head, tail, grep

Versatile Ways To Work With Text Data

In the previous two chapters we have sharpened our skills in creating and managing files and folders, and we are now ready to explore working with data that you find inside of files. In order to understand what we mean by *text data* in this section, let's first discuss the nature of our files stored on our drives. This involves how our computers translate numbers into more useful information to us like text and images.

We are all familiar with counting in the *decimal* system, using combinations of the numbers zero through nine (`0-9`) to represent any number we can imagine, small to large. Computers use a different counting system—the *binary* system—for their internal processing. The binary system only uses two numbers, zero through one (`0-1`), to encode the same numbers we are accustomed to as decimal numbers. For example, what we know as the decimal number `144`, can also be expressed in binary format as `10010000`. Now, visualize flipping a light switch from on to off, and off to on, multiple times. In computers, ones and zeros can correspond to a rapid sequence of *on* (`1`) and *off* (`0`) electrical states. Electricity is the driving force that controls hardware, and computer processors are optimized to handle these ones and zeros efficiently. This is part of the magic behind binary data!

All files are stored as *binary data*, meaning that they are encoded as a set of ones and zeros. We refer to files that contain human-readable content as *text data* files, in that they encode written-language text. Computer scientists have agreed on a few standards that map characters in a language to a corresponding number. For instance, in the *ASCII* standard[1], the English uppercase letter *A* is represented as the decimal number *65* inside of the file. These numbers that represent letters are then stored as binary ones and zeros on disk. Applications like the Terminal shell know how to read the binary-encoded number 65, look up what character it represents in the ASCII character set table, and present us with an English uppercase letter *A* as text on the screen.

Text files are often encoded in one of a few different file formats, including ASCII and UTF-8. You may hear files being referred to as ASCII files (pronounced *as-key*), or Unicode[2] files. The ASCII standard mostly represents American English characters and symbols. Unicode is the international standard that supports characters from languages around the world (UTF-8), including American English as well.

There are also other file formats that are commonly used and are referred to as *binary data* files, meaning that the data are encoded using standards that are not readable as written-language. The binary-encoded bytes of data are certainly readable by programs that understand the format. A good example is an image file format—for instance a Portable Network Graphics (PNG) file—that encodes a grid of numbers that represent a color (a mixture of red [R], green [G], and blue [B]) at the given location in the image. Applications that can read the PNG file format will translate these RGB numbers to pixel colors on your screen when displaying the image. For the purposes of this chapter, we will be focusing on everyday commands that are used to work with text data. Our computers are full of text-based files! But just know, there are hundreds of commands that are useful when working with binary data files as well.

In the following sections, we will explore five commands that allow us to view, combine, sort, preview, and filter text-based files. In particular, we will use a very simple command called `cat` to quickly view the contents of a file directly in the terminal. We'll also use the `cat` command to combine many files together into one file. As we become comfortable with the `cat` command, we'll then discuss the command line concept of *pipes*, which allow us to pass data between multiple commands. This is cosmic magic! We will apply our knowledge of pipes and use the `sort` command to arrange our data in the exact order we are interested in. We'll then learn how to preview the beginning and ends of files using the `head` and `tail` commands, which is amazingly helpful when your files are extremely large, and you want to get a feeling for the contents. We will wrap up the chapter with the `grep` command, which assists us in filtering the lines of text in a file, only returning the exact lines we are interested in.

We will be working with some stellar data in this chapter. And we mean stellar! Our examples revolve around the 175 most prominent stars in the sky, and some fun facts about their locations, distances from us, and their brightness as we see them with the naked eye. It's launch time!

The cat Command—Viewing and Combining Files

While we love cats and dogs alike, we really enjoy the `cat` command for its absolute simplicity. It is a shorthand for the word *catenate* (similar to *concatenate*), and was originally created to combine files

together. However, we'll first use it in its easiest form, which is to just display the contents of a file directly in the terminal window. To do this, let's first have a look at a collection of files that help us illustrate these commands—files that contain lists of stars we see in the night sky. These files are organized by tropical zodiac signs and contain text data describing the most prominent stars aligned with each sign. In *Example 58* we change into the directory containing these files and list them to get a sense of what files we will be working with.

Example 57: Listing the text files representing the twelve tropical zodiac signs and the Sun.

```
chris@ophir ~ % cd examples/chapter-05/signs
chris@ophir signs % ls
Aquarius.txt    Gemini.txt  Sagittarius.txt  Virgo.txt
Aries.txt       Leo.txt     Scorpio.txt
Cancer.txt      Libra.txt   Sun.txt
Capricorn.txt   Pisces.txt  Taurus.txt
```

Great! We see that we have twelve files—plus a thirteenth for the Sun—that are named after the signs of the tropical zodiac, each representing a thirty degree area of the sky along the path of the Sun. Let's take a look inside one of the files using the `cat` command, shown in *Example 59*.

Example 58: Using the `cat` command to view a file's contents in the terminal window on a Mac.

```
chris@ophir constellations % cat Cancer.txt
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
Procyon             26°08'-Cancer       yellow-sub-giant    0011ly  +0.37
Pollux              23°33'-Cancer       orange-giant        0034ly  +1.14
Adhara              21°07'-Cancer       blue-bright-giant   0405ly  +1.50
Castor-A            20°35'-Cancer       white-dwarf         0051ly  +1.58
Wezen               23°45'-Cancer       yellow-super-giant  1600ly  +1.84
Menkalinan          00°15'-Cancer       white-dwarf         0081ly  +1.90
Alhena              09°27'-Cancer       white-sub-giant     0109ly  +1.92
Mirzam              07°32'-Cancer       blue-bright-giant   0493ly  +1.97
Aludra              29°53'-Cancer       blue-super-giant    2000ly  +2.45
Tejat               05°39'-Cancer       red-giant           0232ly  +2.87
Gomeisa             22°32'-Cancer       blue-dwarf          0162ly  +2.89
Castor-B            20°35'-Cancer       white-dwarf         0051ly  +2.90
Tau-Puppis          28°05'-Cancer       orange-giant        0182ly  +2.90
Mebsuta             10°17'-Cancer       yellow-super-giant  0845ly  +2.98
Furud               07°43'-Cancer       blue-dwarf          0362ly  +2.99
```

Awesome! We chose to use the `Cancer.txt` file as a single argument, and the `cat` command read the contents of the file and printed them directly to standard out, displaying the contents directly in the terminal. Since there is only a single file argument, there is no "concatenation" taking place in this example, but rather just the display. This is part of the magic of the `cat` command—it is super simple to see the contents of a file with very little effort!

Now that we know how to view the contents of a file, let's take a minute to understand what we are viewing. Each file shows a list of stars, one per line of the file. The first line of this text file lists the star named *Sirius*, which is a beautiful and colorful star, emitting dominantly white light back to Earth. *Figure 36* shows an image of Sirius and how intensely bright it is, even from a distance of 8 light years away—47 trillion miles or 75 trillion kilometers![3]

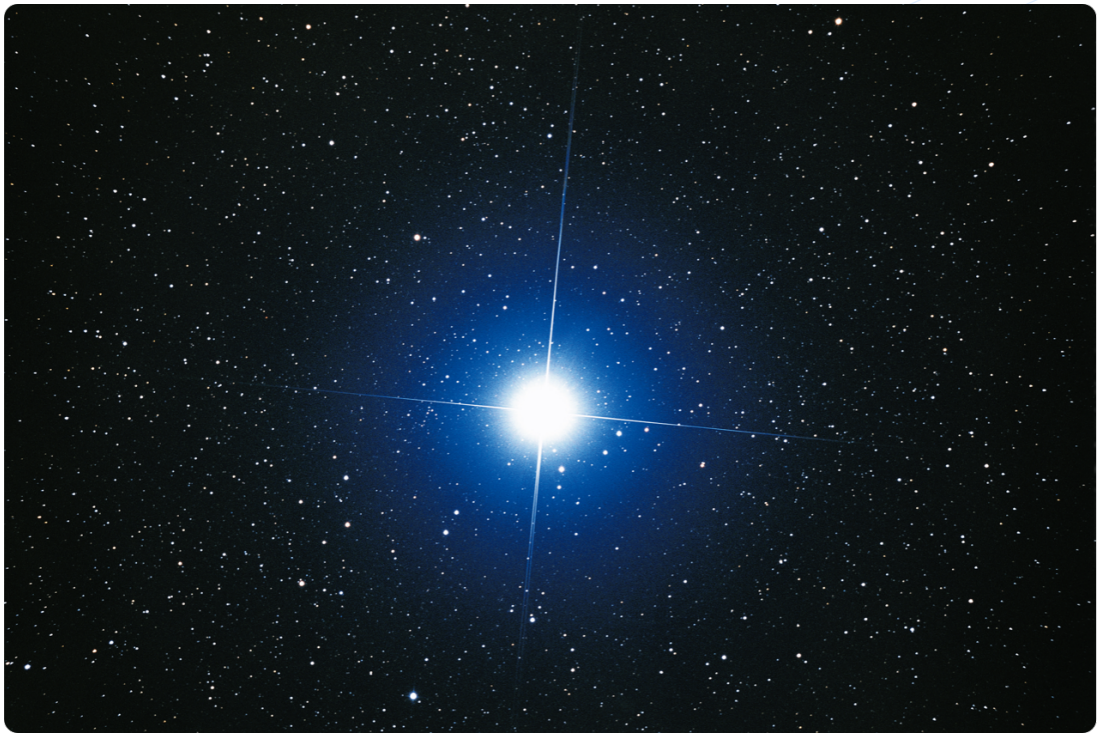

Figure 36. An image of Sirius, a white dwarf star 8 light years away from the Earth! Image credit Akira Fujii.

We're now getting a sense of what the data in these files represent, based on the image in *Figure 36*. It's stellar! Let's dig a little deeper so we understand the information on each line of the file and what each column of data represents. A line of a text file is a single horizontal row, followed by an invisible, non-printing *line ending* character. This is also known as a *record delimiter*.

Text-based data files have a magical framework used for formatting and display that includes invisible characters. These characters provide cues to applications rendering the data. For instance, in ASCII and UTF-8 encodings, there are special non-visible characters in a file that designate a *line ending*. In Unix-like systems (Mac and Linux) it is the *linefeed* character, also know as a *newline* character. It is often written as `LF` or as `\n` when conveying the invisible character. For Windows text files, there are two special characters used as a line ending—a *carriage return* character followed by a *linefeed* character. This is often written as `CRLF` or `\r\n`.

Have a look at *Table 4*, which presents the same data from our `cat` command in *Example 59*.

Table 4. The contents of the `Cancer.txt` file shown as a table with columns and rows, highlighting the orderly structure of the file.

❶ Name	❷ Tropical Sign	❸ Spectral Class	❹ Distance	❺ Brightness
Sirius	14°26'-Cancer	white-dwarf	0008ly	-1.46
Canopus	15°19'-Cancer	white-bright-giant	0309ly	-0.74
Procyon	26°08'-Cancer	yellow-sub-giant	0011ly	+0.37
Pollux	23°33'-Cancer	orange-giant	0034ly	+1.14
Adhara	21°07'-Cancer	blue-bright-giant	0405ly	+1.50
Castor-A	20°35'-Cancer	white-dwarf	0051ly	+1.58
Wezen	23°45'-Cancer	yellow-super-giant	1600ly	+1.84
Menkalinan	00°15'-Cancer	white-dwarf	0081ly	+1.90
Alhena	09°27'-Cancer	white-sub-giant	0109ly	+1.92
Mirzam	07°32'-Cancer	blue-bright-giant	0493ly	+1.97
Aludra	29°53'-Cancer	blue-super-giant	2000ly	+2.45
Tejat	05°39'-Cancer	red-giant	0232ly	+2.87
Gomeisa	22°32'-Cancer	blue-dwarf	0162ly	+2.89
Castor-B	20°35'-Cancer	white-dwarf	0051ly	+2.90
Tau-Puppis	28°05'-Cancer	orange-giant	0182ly	+2.90
Mebsuta	10°17'-Cancer	yellow-super-giant	0845ly	+2.98
Furud	07°43'-Cancer	blue-dwarf	0362ly	+2.99

Table 4 really shows us the structure of the file, which is similar to a spreadsheet. Each line of the file represents a row in the table with characteristics of the star in each column of the table. We see that this file in our dataset contains seventeen of the most prominent stars aligned with the tropical sign of Cancer, one per row.

❶ Column 1 lists the star name.

❷ Column 2 shows the location of the star as degrees and minutes within a tropical sign of the zodiac.

❸ Column 3 of the table, the spectral class, is a short description of the quality of the light being emitted by the star and the relative size of the star.

❹ Column 4 shows how far away the star is in light years (ly).

❺ Column 5 shows the brightness magnitude of the star as we experience them visually from Earth.

Note that brightness is on an inverse scale, so the smaller the number, the brighter the star looks in the sky. Let's compare two of the stars in the sign of Cancer. At `-1.46`, Sirius is considered a "minus-one-magnitude" star (super bright) compared to its celestial companion, Furud. At `2.99`, Furud is a second-magnitude star, and super close to being a third-magnitude star. It is still very bright, but four magnitudes less bright than Sirius, almost five.

Table 4 gives us a good feel for the data in the `Cancer.txt` file, but what about the rest of the files? Let's first combine the `Cancer.txt` file with `Sun.txt` file for a little more comparison. We can do so by passing the two file names as arguments to the `cat` command, as shown in *Example 60*.

Example 59: Using the `cat` command to concatenate the contents of two files and display the results in the terminal window on a Mac.

```
chris@ophir signs % cat Cancer.txt Sun.txt
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
Procyon             26°08'-Cancer       yellow-sub-giant    0011ly  +0.37
Pollux              23°33'-Cancer       orange-giant        0034ly  +1.14
Adhara              21°07'-Cancer       blue-bright-giant   0405ly  +1.50
Castor-A            20°35'-Cancer       white-dwarf         0051ly  +1.58
Wezen               23°45'-Cancer       yellow-super-giant  1600ly  +1.84
Menkalinan          00°15'-Cancer       white-dwarf         0081ly  +1.90
Alhena              09°27'-Cancer       white-sub-giant     0109ly  +1.92
Mirzam              07°32'-Cancer       blue-bright-giant   0493ly  +1.97
```

```
Aludra              29°53'-Cancer       blue-super-giant    2000ly  +2.45
Tejat               05°39'-Cancer       red-giant           0232ly  +2.87
Gomeisa             22°32'-Cancer       blue-dwarf          0162ly  +2.89
Castor-B            20°35'-Cancer       white-dwarf         0051ly  +2.90
Tau-Puppis          28°05'-Cancer       orange-giant        0182ly  +2.90
Mebsuta             10°17'-Cancer       yellow-super-giant  0845ly  +2.98
Furud               07°43'-Cancer       blue-dwarf          0362ly  +2.99
Sun                 Varies              yellow-dwarf        0000ly  -26.7
```

So easy! We've just combined the two files together, sending the results to standard output where it is displayed in the terminal. Notice that the Sun has a visual magnitude of `-26.7`, obviously the brightest star in the sky! Both Sirius and the Sun are considered dwarf stars, and the Sun is as close as it gets! It is listed as zero light years away, but in actuality it takes around eight minutes for the Sun's light to reach us, and it is 93 million miles away, or close to 150 million kilometers. That of course is a huge distance, but as we'll see shortly, some of the stars we see in the sky are so amazingly far away, we have to use our imagination just to comprehend the vastness of the cosmos!

Now that we know how to combine files, let's explore the data and answer some questions— *What are the brightest stars in the sky? In fact, what are the 21 brightest stars? Of the brightest stars, which are the most distant?* To answer these and other questions, let's combine all of the tropical zodiac files together into one big list, and then sort them by visual brightness, and grab the brightest 21 stars. We'll then sort the results by distance, and have a look at the most distant, but super bright stars. They must be really big! To complete our work, we will need the `sort` command and the `head` command, which we will explore. But first we need to learn about command line pipes, which we discuss in the next section. Let's dive into the stars!

Pipes—The Power Of Compound Commands

In *Chapter 3. File Commands*, we learned how to use redirection to send the output of a command into a file using the `>` (greater-than) and `>>` (double-greater-than) operators. In Unix-like systems, there is another similar concept called *pipes*, which allow you to send the output from one command into another command. This is one of the most powerful and magical tools in our toolbox! To do this, we use a different symbol from the keyboard—the `|` (pipe) symbol, which is sometimes referred to as a vertical bar. When we chain commands together using the pipe symbol, collectively it is called a *pipeline*. Have a look at *Figure 37* which shows the parts of a pipeline with three commands.

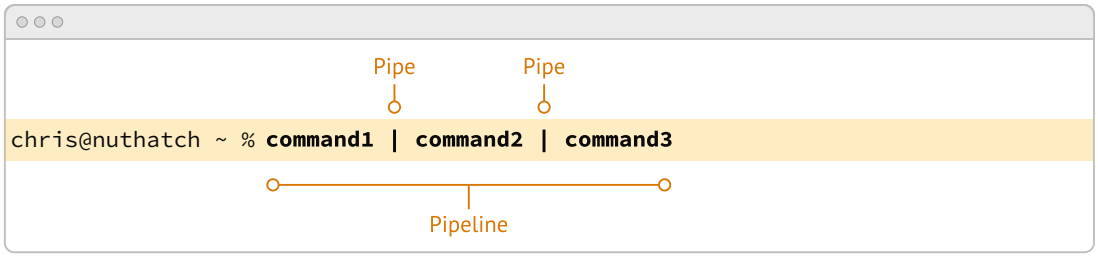

Figure 37. The parts of a command pipeline, where each command is separated by the pipe symbol.

It is super easy, and super powerful! Let's take a moment to understand what is going on. The principle is that you run the first command and send its results to the second command. It in turn processes what it received, and sends its output to the third command. And lastly, the third command processes what it received and produces the final results of the pipeline. For instance in our star data pipeline, you will combine the data files with the `cat` command, and pipe the results to the `sort` command. The results of the sorting will later be piped to the `head` command, where you will filter the sorted list and show just the 21 brightest stars. To see a visual depiction of a pipeline, have a look at *Figure 38*, which shows the data being passed from left to right through the command pipeline.

Figure 38. Visualizing a command pipeline. Command1 (c1) output becomes input for command2 (c2). Command2 output becomes input for command3 (c3). Command3 produces the final output of the pipeline.

This is absolutely amazing! By using pipelines, we can chain together as many commands as we need to get the job done. Let's give this a try in the next section where we concatenate all of our star files organized by tropical sign into a big list, and then sort that list.

The sort Command—Sorting the Contents of a File

In order to answer our questions about what are the brightest stars in the night sky, we'll need to be able to sort our data. And yes, we have the magic! The `sort` command is a super useful tool for reordering data. We can use it as a standalone command by passing in a file name as an argument to sort the contents. However, in our case we are going to use it in a pipeline where we sort the output of the previous `cat` command. Have a look at *Example 61* where we concatenate all of our tropical sign

files that contain star information into one large output, and we send it to the `sort` command with a `|` (pipe) symbol.

Example 60: Using the `cat` command in a command pipeline to concatenate the contents of all text files in a directory and then sort the lines.

```
chris@ophir signs % cat *txt | sort
Achernar            15°39'-Pisces       blue-dwarf          0140ly  +0.46
Acrab               03°32'-Sagittarius  blue-dwarf          0404ly  +2.50
Acrux-A             12°12'-Scorpio      blue-subgiant       0322ly  +1.34
Beta-Arae           24°33'-Sagittarius  orange-super-giant  0646ly  +2.80
Beta-Hydri          01°17'-Aquarius     yellow-sub-giant    0024ly  +2.80
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
Elnath              22°55'-Gemini       blue-giant          0134ly  +1.65
Fang                03°17'-Sagittarius  blue-dwarf          0586ly  +2.91
Gacrux              07°05'-Scorpio      red-giant           0089ly  +1.64
Hadar               24°08'-Scorpio      blue-giant          0392ly  +0.58
Imai                06°01'-Scorpio      blue-sub-giant      0345ly  +2.75
Kakkab              23°51'-Scorpio      blue-giant          0465ly  +2.29
...
Lang-Exster         10°01'-Aquarius     orange-giant        0200ly  +2.85
Mahasim             29°57'-Pisces       white-dwarf         0166ly  +2.62
Naos                18°54'-Leo          blue-super-giant    1100ly  +2.25
Okab                20°08'-Capricorn    white-dwarf         0083ly  +2.99
Paikauhale          11°48'-Sagittarius  blue-dwarf          0474ly  +2.81
Rasalhague          22°47'-Sagittarius  white-giant         0049ly  +2.07
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Sun                 Varies              yellow-dwarf        0000ly  -26.7
Tarazed             01°16'-Aquarius     orange-bright-giant 0395ly  +2.72
Unukalhai           22°25'-Scorpio      orange-giant        0074ly  +2.63
Vathorz-Posterior   29°11'-Libra        blue-dwarf          0456ly  +2.84
Wezen               23°45'-Cancer       yellow-super-giant  1600ly  +1.84
Xamididamura        16°30'-Sagittarius  blue-dwarf          0501ly  +3.00
Yed-Prior           02°38'-Sagittarius  red-giant           0171ly  +2.75
Zaurak              24°13'-Taurus       red-giant           0203ly  +2.94
```

Wow! Sorting is that easy! Try it for yourself, since the output in *Example 61* was truncated for display purposes. We see that when we use the `cat` command and pass it all of the thirteen files ending in `txt` (using a `*` (star) wildcard to expand the file names), there are 175 lines of output, one line for each star. We then pipe the results from the `cat` command into the `sort` command, and the final result is a

listing of all lines within those files, sorted with the default sort criteria. Notice that the results printed to the terminal screen are in the default *alphabetical* sort order, based on the text content of each line. This doesn't quite help us with our question about which stars are the brightest, so let's discuss sorting data numerically rather that alphabetically.

Sorting numerically

We're making progress! That was your first successful command pipeline! You can now modify your pipeline to sort the lines of text data in other ways than just alphabetical order. Have a look at the manual page for the `sort` command to get a feeling for the command options. One of the indispensable options for sorting text data as columns is the `-k` option, which lets us tell the command which field (column, or key) to use to sort the data. Let's use this option to sort our star data, first by magnitude (brightness).

Take a look at *Example 61* and notice how the lines of data have blank spaces, and the data are visually arranged as columns and rows, as we highlighted in *Table 4*. The brightness numbers are in the fifth column, which is also referred to as the fifth *field*. The manual page tells us that the `-k` option takes one or more field numbers to sort by, and in our case we will use `-k 5` to sort by the fifth column. We'll also use the `-g` option to do a "general numeric sort". By using this option, the command will know how to handle the `+` (plus) and `-` (minus) signs the prefix our brightness values. Go ahead and give it a try! *Example 62* shows the results of our modified command pipeline.

Example 61: Combining all star files using the `cat` *command and piping the results to the* `sort` *command, sorting numerically on column 5.*

```
chris@ophir signs % cat *txt | sort -k 5 -g
Sun                 Varies              yellow-dwarf        0000ly  -26.7
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
Arcturus            24°34'-Libra        orange-giant        0037ly  -0.05
Rigil-Kentaurus     29°47'-Scorpio      yellow-dwarf        0004ly  +0.01
...
Pollux              23°33'-Cancer       orange-giant        0034ly  +1.14
Fomalhaut           04°12'-Pisces       white-dwarf         0025ly  +1.16
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
...
Diphda              02°56'-Aries        yellow-giant        0096ly  +2.00
Hamal               08°00'-Taurus       orange-giant        0066ly  +2.01
Polaris             28°55'-Gemini       yellow-super-giant  0433ly  +2.02
Menkent             12°39'-Scorpio      orange-giant        0059ly  +2.05
```

```
...
Delta-Centauri      27°29'-Libra        blue-sub-giant      0415ly  +2.42
Sabik               18°18'-Sagittarius  white-dwarf         0088ly  +2.42
Scheat              29°22'-Pisces       red-giant           0196ly  +2.42
...
Yed-Prior           02°38'-Sagittarius  red-giant           0171ly  +2.75
Zubenelgenubi       15°25'-Scorpio      white-sub-giant     0076ly  +2.75
Beta-Lupi           25°01'-Scorpio      blue-giant          0383ly  +2.76
...
Castor-B            20°35'-Cancer       white-dwarf         0051ly  +2.90
Tau-Puppis          28°05'-Cancer       orange-giant        0182ly  +2.90
Fang                03°17'-Sagittarius  blue-dwarf          0586ly  +2.91
...
Beta-Triangulum     12°42'-Taurus       white-sub-giant     0127ly  +3.00
Vathorz-Prior       23°14'-Libra        blue-super-giant    1400ly  +3.00
Xamididamura        16°30'-Sagittarius  blue-dwarf          0501ly  +3.00
```

Perfect! Just like that, you just did your first command line sort! All 175 stars were displayed in the terminal, sorted based on their numeric brightness value in column five of the data. For display purposes, the results in *Example 62* have been truncated, but your results should be the fully sorted list of stars. The list scrolls out of view quickly, so it's a challenge to see the brightest stars without scrolling up in the Terminal window, so let's go one step further in our pipeline and zone in on the top 21 brightest stars using the `head` command—because 21 is just an awesome number!

The head Command—Previewing the Top of a File

We will continue with some handy `sort` command features shortly, but let's turn our attention to the `head` command, so we can answer our question— *What are the 21 brightest stars in the sky?* The `head` command is a very simple text handling command that helps you display just the lines at the top of a file, and by default it will show you the top 10 lines. Conveniently, with the `-n` option, we can specify the exact number of lines we're interested in. So go ahead and add the `head` command to your previous star exploration pipeline, and let's find those top 21 stars! *Example 63* shows the results of our pipeline.

Example 62: Displaying the top 21 lines of text from a command pipeline using the `head` *command.*

```
chris@ophir signs % cat *txt | sort -k 5 -g | head -n 21
Sun                 Varies              yellow-dwarf        0000ly  -26.7
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
```

```
Arcturus            24°34'-Libra        orange-giant        0037ly  -0.05
Rigil-Kentaurus     29°47'-Scorpio      yellow-dwarf        0004ly  +0.01
Vega                15°39'-Capricorn    white-dwarf         0025ly  +0.03
Capella             22°12'-Gemini       yellow-giant        0043ly  +0.08
Rigel               17°11'-Gemini       blue-super-giant    0860ly  +0.13
Procyon             26°08'-Cancer       yellow-sub-giant    0011ly  +0.37
Betelgeuse          29°06'-Gemini       red-super-giant     0498ly  +0.42
Achernar            15°39'-Pisces       blue-dwarf          0140ly  +0.46
Hadar               24°08'-Scorpio      blue-giant          0392ly  +0.58
Altair              02°07'-Aquarius     white-dwarf         0017ly  +0.76
Aldebaran           10°08'-Gemini       orange-giant        0067ly  +0.86
Antares             10°06'-Sagittarius  red-super-giant     0554ly  +0.91
Spica               24°11'-Libra        blue-subgiant       0250ly  +0.97
Pollux              23°33'-Cancer       orange-giant        0034ly  +1.14
Fomalhaut           04°12'-Pisces       white-dwarf         0025ly  +1.16
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
Mimosa              11°59'-Scorpio      blue-giant          0279ly  +1.25
Toliman             29°47'-Scorpio      orange-dwarf        0004ly  +1.33
```

Ah, such tidy results! By adding the `head` command to the pipeline, you just trimmed the results down to the top 21 lines of the output, zoning in on the brightest stars in the sky! And have a look at *Example 63*, where the star Sirius, which aligns with the sign of Cancer, is the brightest star in the sky, second only of course to our very own Sun. So cool! We mentioned earlier in the chapter that Sirius is 8 light years away from Earth, or 47 trillion miles or 75 trillion kilometers. Definitely take a look at Sirius as it rises in the eastern sky at night. With a pair of binoculars or a telescope, you can see its colors dancing around like a disco ball!

With a few short commands, we are seeing how easy it is to combine files, sort the contents, and extract some very enlightening information from the results. What is even better is that these commands work equally well with files that have millions of lines of text, an enormous benefit from these small tools already present on your computer!

But let's now continue exploring our commands and the stars, because we have more intriguing questions to answer. In *Example 63* we see Canopus, the third brightest star in the sky, with only Sirius and the Sun being brighter. Canopus is the brightest star in the southern hemisphere, and known as the Great Star of the South, it is 309 light years away from the Earth. Wow! That equates to approximately 1.8 quadrillion miles or 2.9 quadrillion kilometers. Wait, what?! That is very far away. And despite this distance, Canopus shines brightly in the night sky, particularly for everyone in the southern hemisphere. Its diameter is over 70 times greater than our Sun, and it is classified as a white

bright giant.

So this is interesting—we have some very bright, but very distant stars in our list of the 21 brightest stars, but *which is the most distant of these very bright stars in the sky*? Now that we know how to sort by columns in a text file, we can answer this question by sorting by distance in our command pipeline. But first, to get a better understanding of how all of our commands can recognize one column from the next, let's discuss the magical framework of both visible and invisible formatting character that are used to structure text data files.

Formatting characters in text data

When we glance at the lines of text in our previous *Example 63*, it's very easy for humans to see that the data are not only organized as lines (i.e. records or rows), but each line is also organized into vertical collections (i.e. fields or columns). *Table 4* highlights this structure. But how do commands like `sort` process the files as rows and columns so readily?

Text data files use a variety of both visible and invisible characters to format the contents, and collectively they provide a subtle yet powerful framework for organizing data. When you create text files, you can also use these characters to keep things organized.

As we discussed before, we know that the lines of star data in our text files have an invisible `\n` (newline) line ending character, which defines what a row is in the data table. You may be asking—*What defines each of the columns?* Part of the magical framework for formatting and display of text data includes a *field separator*. Field separators are characters in the file that separate the text on a given line into fields (columns). These characters can be anything, but are often comma, tab, or space characters. Have a look at *Figure 39*, which depicts text files with both tabs and commas as field separators (column separators).

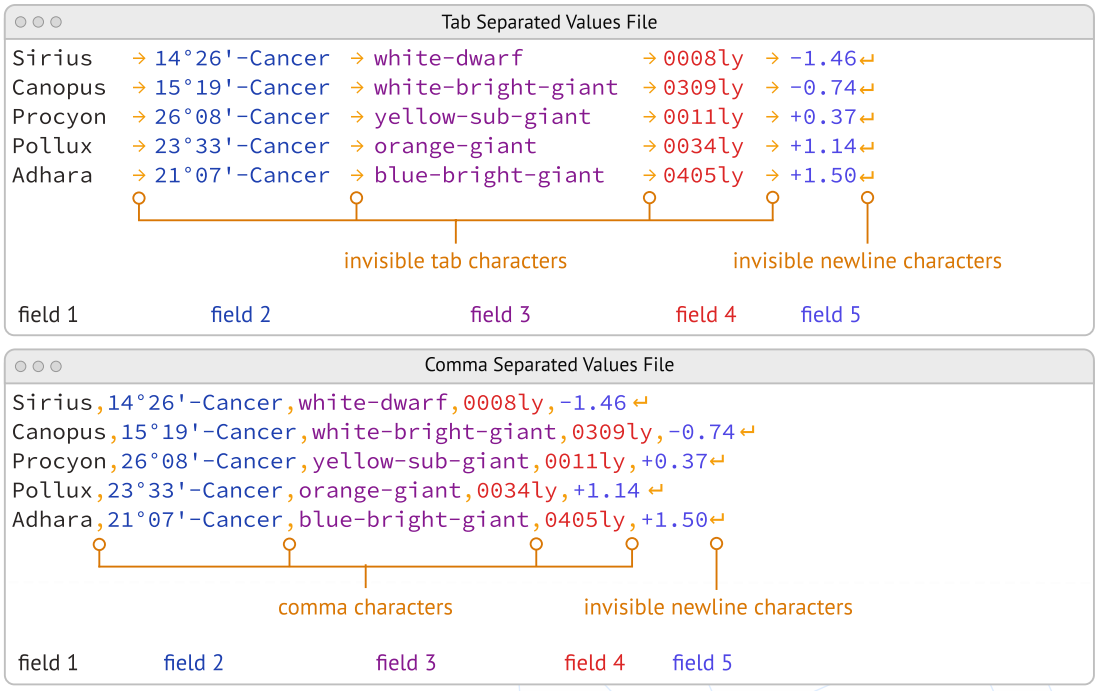

Figure 39. Visualizing text file fields (columns) in two types of files. Tab separated values (TSV) files use invisible tab characters between columns as a field separator. Comma separated values (CSV) files use commas between columns as a field separator. Both use invisible newline characters as the record delimiter (line ending).

The text is colored in *Figure 39* to highlight the vertical columns of information. Notice that in the top example the field separator is the tab keyboard character. This is an invisible (non-printing) character that is used for indentation. So *tab separated values* files, or TSV files as a shorthand, have a tab character in between columns of data. These file names often end in `.tsv` or `.txt`. TSV files can be directly imported into spreadsheet applications.

The second example in *Figure 39* shows a `,` (comma) field separator. Text data files with columns separated by a comma are often referred to as *comma separated values* files, or "CSV" files as a shorthand. These file names often end in `.csv`, and can also be directly imported into spreadsheet applications.

When viewing text files in the terminal application, the width of the terminal window may affect the display. If the window is too narrow, lines of data will wrap to the next line, causing the data to look messy. In this case, increase the width of your terminal application window, and the columns of text files should align appropriately.

Sorting files based on field separators

Given this knowledge about formatting characters used in text data files, let's continue to sort our star data based on each star's distance from us in light years. Many commands take advantage of these invisible formatting characters when handling text files, and the `sort` command is one of them! When you look at the manual page for the `sort` command, you will see that the `-t` option lets you tell the command what character to use to separate fields (columns). As we mentioned earlier, our star data files use the invisible tab character to separate the columns, so we need to tell the sort command to use a tab as the field separator. Let's give this a try!

In our second sort command, we will use the `-k 4` option to specify the fourth field to sort by (distance). We will sort numerically using the `-n` option, and will use the `-t` option to explicitly set the field separator.

The command pipeline shown below includes the `-t` option followed by two consecutive double-quotes. The command is almost complete, but *how do we insert a tab character between the double-quotes?* When you are typing this command and press the `Tab` key to insert a tab character in that location, the shell interprets the keypress as a desire to use the tab completion feature of the shell, which we don't want. We want to insert an invisible tab character. *How do we solve this*? There is a little gem of a key combination that lets us insert a tab character on the command line.

```
cat *txt | sort -k 5 -g | head -n 21 | sort -k 4 -n -t "" ❶
```

❶ Move your cursor directly after the first `"` (double-quote) character—the block cursor will be sitting on the second `"` (double-quote) character. Then type the following key combination to insert a tab character— `Control` + `v` then `Tab`. This tells the shell to forego tab completion, and just insert the invisible character. Ah, the magic!

```
cat *txt | sort -k 5 -g | head -n 21 | sort -k 4 -n -t "    "
```

We now have an invisible tab character between our double-quotes and are ready to sort our data! Take a look at *Example 64*, which concatenates all of the star files and sorts them by visual brightness, grabs the top 21 lines, and then sorts those lines numerically by distance in light years with an explicit tab character field separator.

Example 63: Sorting the output of the `head` *command using a tab as the field separator (* `-t "	"` *) and the fourth field as the column key (* `-k 4` *) numerically (* `-n` *). The example shows uses a* `\` *(backslash) character to create multi-line command for display purposes.*

```
chris@ophir signs % cat *txt | sort -k 5 -g | \
head -n 21 | sort -k 4 -n -t "	" ❶
Sun                 Varies              yellow-dwarf        0000ly  -26.7
Rigil-Kentaurus     29°47'-Scorpio      yellow-dwarf        0004ly  +0.01
Toliman             29°47'-Scorpio      orange-dwarf        0004ly  +1.33
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Procyon             26°08'-Cancer       yellow-sub-giant    0011ly  +0.37
Altair              02°07'-Aquarius     white-dwarf         0017ly  +0.76
Fomalhaut           04°12'-Pisces       white-dwarf         0025ly  +1.16
Vega                15°39'-Capricorn    white-dwarf         0025ly  +0.03
Pollux              23°33'-Cancer       orange-giant        0034ly  +1.14
Arcturus            24°34'-Libra        orange-giant        0037ly  -0.05
Capella             22°12'-Gemini       yellow-giant        0043ly  +0.08
Aldebaran           10°08'-Gemini       orange-giant        0067ly  +0.86
Achernar            15°39'-Pisces       blue-dwarf          0140ly  +0.46
Spica               24°11'-Libra        blue-subgiant       0250ly  +0.97
Mimosa              11°59'-Scorpio      blue-giant          0279ly  +1.25
Canopus             15°19'-Cancer       white-bright-giant  0309ly  -0.74
Hadar               24°08'-Scorpio      blue-giant          0392ly  +0.58
Betelgeuse          29°06'-Gemini       red-super-giant     0498ly  +0.42
Antares             10°06'-Sagittarius  red-super-giant     0554ly  +0.91
Rigel               17°11'-Gemini       blue-super-giant    0860ly  +0.13
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
```

❶ A quoted invisible tab character is used as the `-t` option value.

Excellent! You just completed your first command pipeline with multiple sorts! The possibilities are endless when creating pipelines, and we can see that our results have reordered the 21 brightest stars in ascending order according to the fourth field—the distance in light years from Earth. You can see that the last four stars in the list are Betelgeuse, Antares, Rigel, and Deneb, and they are all categorized as super-giant stars. Deneb, the brightest star in the constellation of Cygnus (the Swan), is therefore the most distant of the 21 brightest stars. At an estimated 2600 light years away from Earth, Deneb is a massive and luminous star, approximately 200 times the diameter of our Sun![4] Now we can see that these very distant stars—with Deneb being the equivalent of 15 quadrillion miles away (24 quadrillion kilometers)--are among the brightest stars in the sky because they are so incredibly large! By using a few simple commands within a command pipeline, we are able to answer some interesting questions about our star data. We discovered that Deneb is the most distant of the brightest 21 stars. However, *of all*

175 brightest stars in the sky, which is the most distant? Let's learn about the `tail` command and answer this question.

The tail Command—Previewing the Bottom of a File

We learned about the `head` command in a previous section, which displays the top of a file. The `tail` command is very similar, but displays the end of a file, with a default value of ten lines. This command is super useful when your command results are expected to be extremely long, and you are only interested in the last lines of the output.

In our case, let's change our command pipeline where we sort the entire 175-star list using the distance field (column 4), and then return only the 10 most distant stars using the `tail` command. Our command pipeline will be similar to *Example 64*, but a bit shorter:

```
cat *txt | sort -k 4 -n -t "    " | tail
```

Go ahead and try this command yourself, and remember to use the handy trick for inserting an invisible tab character on the command line— `Control` + `v` then `Tab`. *Example 65* shows the results of this command.

Example 64: Using the `tail` command to limit command output to the last ten lines. All of the tropical sign star files are combined, sorted numerically by distance in light years, and filtered to show just the last ten lines of the result.

```
chris@ophir signs % cat *txt | sort -k 4 -n -t "    " | tail
Almaaz              19°11'-Gemini       yellow-super-giant  1400ly  +2.99
Vathorz-Prior       23°14'-Libra        blue-super-giant    1400ly  +3.00
Wezen               23°45'-Cancer       yellow-super-giant  1600ly  +1.84
Sadr                25°10'-Aquarius     yellow-super-giant  1800ly  +2.23
Girtab              27°52'-Sagittarius  blue-giant          1930ly  +2.99
Alnilam             23°49'-Gemini       blue-super-giant    2000ly  +1.69
Aludra              29°53'-Cancer       blue-super-giant    2000ly  +2.45
Arneb               21°44'-Gemini       yellow-super-giant  2200ly  +2.57
Hatysa              23°21'-Gemini       blue-white-giant    2300ly  +2.77
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
```

So easy! The `tail` command is a go-to utility for grabbing the last lines of your output, and is particularly useful when the results are thousands of lines long and you are only interested in the last

lines. Another very common scenario for the tail command is to view the most recent lines being appended to log files on your computer.

Computers applications and other processes running on your system regularly create files that are called *logs*. These files are used to periodically append information about the running application, like status information, warnings, errors, and crashes. These files often have a file extension of `.log`, for instance `system.log`, and are used for troubleshooting.

In command line lingo, we use the phrase "tail a file", which means to use the `tail` command with the `-f` option. This is another gem of a tool where you can monitor what is being added into a file in real time. Let's set up a small example of how to use the `tail -f` option. To do so, we'll need two terminal windows open—one to write lines to a file, and another to tail the file. Choose which operating system you are using below for a reminder on opening new terminal windows.

Mac

(1) In the Terminal application, press the keyboard shortcut `Command` + `N`; or (2) Choose the Terminal application's *Shell* > *New Window* > *New Window with Profile - Basic* menu item.

Linux

(1) In the Gnome Terminal application, press the keyboard shortcut `Command` + `Shift` + `N`; or (2) Click on the Gnome Terminal application's menu icon in the window title bar and choose *New Window*.

Windows

(1) In the Windows Terminal application, hold the the the keyboard + `Shift` button and click on the `+` (plus) icon in the window title bar; or (2) Click on the Windows Terminal application's dropdown menu in the window's title bar, hold down the `Shift` button, and choose the *Ubuntu* menu item.

Each of the graphical steps are shown in the screenshots in *Figure 40*.

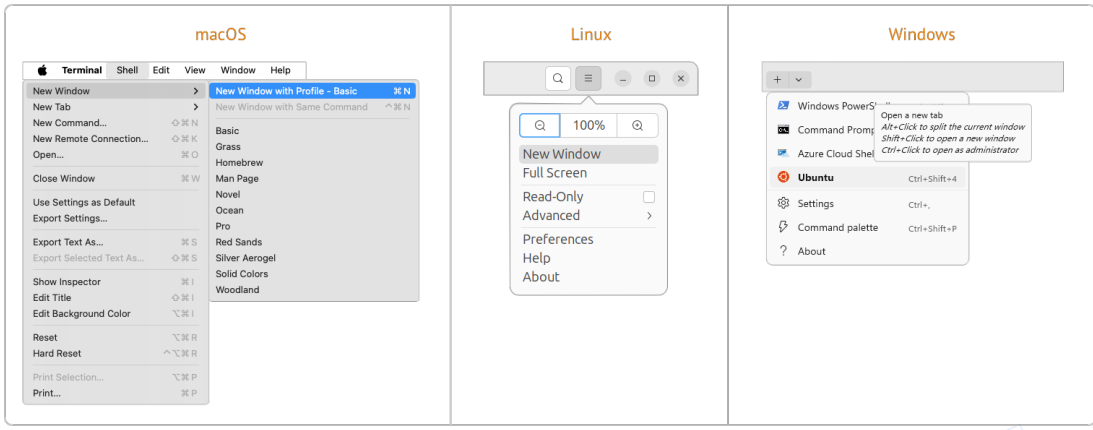

Figure 40. Opening a new terminal window on Mac, Linux, and Windows, using each application's menu dropdown items.

Once you have two terminal windows open, arrange them side-by-side. In the left terminal, issue the following command to redirect a line of text into a file called `stars.log`:

```
chris@ophir ~ % echo "The stars are aligning!" >> stars.log
```

Next, in the right terminal window, use the `tail` command with the `-f` option to view the `stars.log` file in real time:

```
chris@ophir ~ % tail -f stars.log
```

Finally, return to your left terminal window, and repeatedly re-issue the same `echo` command as before, three or four times:

```
chris@ophir ~ % echo "The stars are aligning!" >> stars.log
chris@ophir ~ % echo "The stars are aligning!" >> stars.log
chris@ophir ~ % echo "The stars are aligning!" >> stars.log
chris@ophir ~ % echo "The stars are aligning!" >> stars.log
```

As you do, you'll see each of the lines of text show up in the right terminal window in real time. Your terminals should look similar to what you see in *Figure 41*, which shows the results of tailing a file.

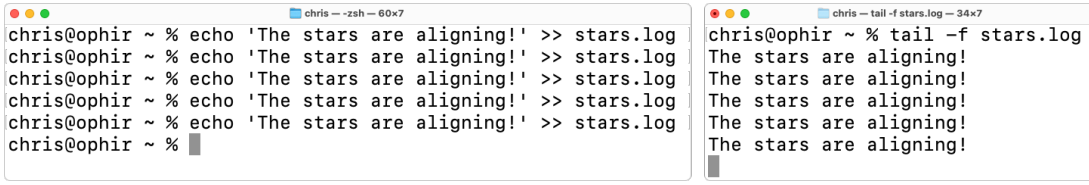

Figure 41. Using two terminal windows to demonstrate how to tail a file. The left window echoes lines of text into a file. The right window displays the content being appended in real time using `tail -f`.

This is a very simple example that demonstrates how the `tail` command can show what is being appended into a file, and shows how easy it is to monitor log files. Both the `head` and the `tail` commands are useful for getting to information at the top and the bottom of files respectively. You may be wondering— *How do you display lines from anywhere in a file?* In the next section we will explore the `grep` command, which allows us to search all lines in a file, and find exactly what we're looking for. Let's have a look!

The grep Command—Filtering Data

As we've been learning, working with text files often involves line-based processing of the contents of a file. This is very much the case when you want to search within a file or many files all at once. To grab exactly what you want from inside of files, the `grep` command is your friend! Go ahead and have a look at the manual page for `grep`. You'll see that it will "print lines that match patterns". When we talk about using `grep` on the command line, it means we are searching for text-based patterns in one or more files, so the arguments you give to the command include at least those two things—a text pattern to search for, and a set of files to search within. Let's begin with a common example where you want to see if a word is inside of a file. *Example 66* shows how to do a simple search within the `Virgo.txt` file for the word `super`. Try this search for yourself!

Example 65: Using the `grep` *command to search within a file for a single word pattern.*

```
chris@ophir signs % grep super Virgo.txt
Suhail              11°32'-Virgo        orange-super-giant  0545ly  +2.21
```

Perfect! You found one line of text in the `Virgo.txt` file that contains the word `super`. So in this case, the pattern we're looking for is the word "super" (a sequence of characters), and the file set we are searching is just the Virgo.txt file. So easy! We see that the star named Suhail is an orange super giant that is 545 light years away from us. Very cool!

The `grep` command is awesome for filtering large amounts of information to find just what you

want. At times, though, you may know what you *don't want* to find more than you know what you *want*. In this case, the `grep` command can exclude the lines you don't want to see by using the `-v` option. This is a handy tool to view lines of a file that you may not even know are present. For example, let's search the star files, and exclude lines that have the word `giant` in them. Take a look at *Example 67*, which excludes giant stars from the list.

Example 66: Using the `grep` *command to search files by excluding lines containg a word pattern with the* `-v` *option.*

```
chris@ophir signs % grep -h -v giant *txt
Altair              02°07'-Aquarius     white-dwarf         0017ly  +0.76
Alderamin           13°07'-Aries        white-dwarf         0049ly  +2.46
Sirius              14°26'-Cancer       white-dwarf         0008ly  -1.46
Castor-A            20°35'-Cancer       white-dwarf         0051ly  +1.58
Menkalinan          00°15'-Cancer       white-dwarf         0081ly  +1.90
Gomeisa             22°32'-Cancer       blue-dwarf          0162ly  +2.89
Castor-B            20°35'-Cancer       white-dwarf         0051ly  +2.90
Furud               07°43'-Cancer       blue-dwarf          0362ly  +2.99
Vega                15°39'-Capricorn    white-dwarf         0025ly  +0.03
Nunki               12°43'-Capricorn    blue-dwarf          0228ly  +2.07
Okab                20°08'-Capricorn    white-dwarf         0083ly  +2.99
Adid-Australis      06°01'-Gemini       blue-dwarf          0612ly  +2.89
Regor               27°21'-Leo          wolf-rayet          1100ly  +1.83
Merak               19°47'-Leo          white-dwarf         0080ly  +2.37
Porrima             10°30'-Libra        bright-yellow-dwarf 0038ly  +2.74
Vathorz-Posterior   29°11'-Libra        blue-dwarf          0456ly  +2.84
Algorab             13°48'-Libra        blue-dwarf          0086ly  +2.94
Achernar            15°39'-Pisces       blue-dwarf          0140ly  +0.46
Fomalhaut           04°12'-Pisces       white-dwarf         0025ly  +1.16
Markab              23°49'-Pisces       blue-dwarf          0133ly  +2.48
Mahasim             29°57'-Pisces       white-dwarf         0166ly  +2.62
Sabik               18°18'-Sagittarius  white-dwarf         0088ly  +2.42
Acrab               03°32'-Sagittarius  blue-dwarf          0404ly  +2.50
Saik                09°34'-Sagittarius  blue-dwarf          0366ly  +2.56
Alpha-Arae          25°17'-Sagittarius  blue-dwarf          0267ly  +2.80
Paikauhale          11°48'-Sagittarius  blue-dwarf          0474ly  +2.81
Gamma-Tri-Australis 09°44'-Sagittarius  white-dwarf         0184ly  +2.85
Alniyat             08°09'-Sagittarius  blue-dwarf          0697ly  +2.89
Fang                03°17'-Sagittarius  blue-dwarf          0586ly  +2.91
Xamididamura        16°30'-Sagittarius  blue-dwarf          0501ly  +3.00
Rigil-Kentaurus     29°47'-Scorpio      yellow-dwarf        0004ly  +0.01
Toliman             29°47'-Scorpio      orange-dwarf        0004ly  +1.33
```

```
Alphecca            12°38'-Scorpio      white-dwarf          0075ly  +2.24
Eta-Centauri        20°14'-Scorpio      blue-dwarf           0306ly  +2.31
Zubeneschamali      19°43'-Scorpio      blue-dwarf           0185ly  +2.62
Iota-Centauri       03°28'-Scorpio      white-dwarf          0058ly  +2.73
Sun                 Varies              yellow-dwarf         0000ly  -26.7
Algol               26°31'-Taurus       blue-dwarf           0090ly  +2.12
Sheratan            04°19'-Taurus       white-dwarf          0059ly  +2.65
Regulus             00°10'-Virgo        blue-dwarf           0079ly  +1.40
Alioth              09°17'-Virgo        white-dwarf          0083ly  +1.77
Alsephina           18°57'-Virgo        white-dwarf          0081ly  +1.77
Alkaid              27°17'-Virgo        blue-dwarf           0104ly  +1.86
Denebola            21°57'-Virgo        white-dwarf          0036ly  +2.13
Mizar               16°03'-Virgo        white-dwarf          0081ly  +2.22
Phecda              00°50'-Virgo        white-dwarf          0083ly  +2.44
Zosma               11°40'-Virgo        white-dwarf          0058ly  +2.53
Cor-Caroli          24°55'-Virgo        bright-yellow-dwarf  0107ly  +2.88
```

Interesting! Forty-eight of the brightest stars are labeled as non-giant stars, and we see many dwarf size stars in this list. It's also very common to pipe the results of a `grep` command into another `grep` command for further filtering. For instance, let's also filter out any dwarf stars from the resulting list, and see what we get. Have a look at the results in *Example 68*.

Example 67: Piping the output of a `grep` *command to another* `grep` *command, continuing to filter with the* `-v` *option.*

```
chris@ophir signs % grep -h -v giant *txt | grep -v dwarf
Regor               27°21'-Leo          wolf-rayet           1100ly  +1.83
```

Ah hah! By filtering out lines that we are not interested in using the `-v` option, we have discovered that a single star, Regor, is not labeled as a giant or dwarf star, but rather it is classified as a "wolf-rayet" star. Amazing! This is like finding a needle in a hay stack when a lot of data are involved, and it demonstrates the power of the `grep` command. Wolf-Rayet stars are very unique in that they are some of the hottest stars that have been observed, with a surface temperature above 50,000 degrees Kelvin. Regor actually is a massive star, with the qualities of a blue giant. Its carbon-rich core is said to cause Wolf-Rayet stars to burn intensely during a relatively short lifespan, and then they are thought to collapse and explode into massive nebulae. *Figure 42* shows Regor—such an amazing star!

Figure 42. An image of Regor, a highly luminous Wolf-Rayet class star 1100 light years away. Image credit: sky-map.org.

Using pattern matching in commands

In the last few examples, we have searched for a pattern that have been single words—like *giant* or *dwarf*. When searching within text files, there are times when we want to find multiple different values that match a pattern rather than a single word (which is a sequence of characters, called a string). Yes, we have the magic to do this! Rather than a single word for our search pattern, we will use what is called a *regular expression* in computing. Regular expressions are little recipes of special characters and rules that define a pattern to search for. Recall in *Chapter 4. Folder Commands* where we learned about path expansion. We used special characters like `{}` (curly braces) and `[]` (square brackets) to tell the shell we want a list or a sequence to be expanded. Regular expressions are similar—they use special characters and some simple rules to help you define a search pattern that matches the sequence of characters you are looking for in the text. An quick example would be the pattern `[0-9]`, which is a little shorthand gem that matches a range of numbers from zero to nine. The `grep` command supports modern regular expressions using the `-E` command option.

In computing, regular expressions are used in most all modern programming languages to match text-based patterns of letters, numbers, and symbols. Learning how to use regular expressions in command line commands is a skill that can be transferred over when learning to program. The term is often shortened to *regex*.

Let's start with an example using the `grep` command, where we can construct our previous command to filter out both dwarf and giant stars in a more concise way. One of the special characters used in regular expression patterns is the `|` (pipe) symbol. When used for pattern matching, the pipe symbol indicates an *OR operator* —meaning that we can search for *pattern1 OR pattern2 OR pattern3*, etc. Let's give this a try with the `-E` option for the `grep` command. Also note that the `|` (pipe) symbol is also a special symbol for commands as we learned earlier—therefore we will wrap our pattern in `""` (double-quote) characters to shield it from being interpreted as a command line pipe. Have a look at *Example 69*, showing the use of a regular expression in the `grep` command.

Example 68: Filtering text data using a regular expression pattern with the `-E` *option of the* `grep` *command.*

```
chris@ophir signs % grep -h -v -E "dwarf|giant" *txt
Regor               27°21'-Leo          wolf-rayet          1100ly  +1.83
```

Excellent! You just searched for two different patterns at the same time using this particular matching syntax of `"dwarf|giant"` as the option value. With the `-E` option, the `grep` command knew to treat the option value as a regular expression and split the two search terms apart based on the `|` (pipe) separator. Using the `-v` option we learned about previously, the `grep` command excluded these terms from the results, and returned the same Wolf-Rayet star—Regor—as before, and this command is more concise than the command in *Example 68*. Magical!

Regular expressions can be used in many commands, not just `grep`. We've seen two examples of pattern matching syntax so far—number ranges with brackets like `[0-9]` and the OR operator using a pipe symbol like `"pattern1|pattern2"`. There are many more ways to match all sorts of letters, numbers, invisible characters, words, repeated patterns, and others. Have a look at *Appendix C: Understanding Regular Expressions* to learn about many of the common matching techniques and rules that help you find exactly what you are looking for!

Command Line Data Handling is Awesome!

Wow! We've covered a lot of ground when it comes to handling text data files and the sheer vastness of the cosmos! We've learned that all files are are stored and processed as binary data in computer systems, and that text files store human-readable written language by additionally encoding them the text with agreed-upon standards such as ASCII and UTF-8. We are now familiar with five awesome commands that help us work with text files, and understand how to handle files—both small and large—with just a few command line options. We've acquainted ourselves with the `cat` command which lets us quickly view files directly in the terminal window, and helps us combine potentially thousands of files together effortlessly. In order to send the results of one command into another we learned about the concepts of pipes and pipelines, which are magical command line tools that open up a whole new world of flexibility for processing lines of data. With these tools in our toolbox, we learned to use the `sort` command to order the results of our previous commands in multiple ways, including alphabetically and numerically. We explored the concepts around text file formatting and how both visible and invisible characters are used to separate lines of a file with record delimiters, and columns of a file with field separators—all to keep our data tidy! With the `head` and `tail` commands, we discovered how easy it is to view the tops and bottoms of data files, which helped us zone in on the brightest and most distant stars that we see in the sky. And to round out our text data handling techniques, we familiarized ourselves with the awesome search capabilities of the `grep` command. With an introduction to matching patterns using regular expressions, we see how flexible it is to filter large datasets to find the gems. Grab just the data you want, and only what you want! There's more to explore with regular expressions in *Appendix C. Understanding Regular Expressions*, but let's next turn to some super handy utility commands that make our command line life as convenient as possible. Onward!

[1] ASCII is an acronym that stands for the American Standard Code for Information Interchange. For more details, see Mackenzie, Charles E.. Coded Character Sets: History and Development. Netherlands: Addison-Wesley Publishing Company, 1980. ISBN-13: 9780201144604

[2] The term Unicode refers to the international *Unicode Standard* for encoding text. For details, see Korpela, Jukka K.. Unicode Explained. Germany: O'Reilly Media,2006. ISBN-13: 9780596101213

[3] The term *light year* is a measure of distance, defined by how long it takes light to travel in a one-year period. This equates to 5.88 trillion miles, or 9.46 trillion kilometers. Wow! For details, see Seidelmann, P. Kenneth. Explanatory Supplement to the Astronomical Almanac. United Kingdom: University Science Books, 1992. ISBN-13: 9781891389856

[4] For the physical measurements of Deneb, see the work by Schiller, F., Pryzbilla, N. "Quantitative spectroscopy of Deneb ". Astronomy and Astrophysics Volume 479, Number 3 (March 1 2008): 849-858. https://doi.org/10.1051/0004-6361:20078590

Utility Commands

less, history, open

Utilities That Make Your Life Easier

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

The less Command—Paging Output for Easy Viewing

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

The history Command—Using Your Command History

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

The open Command—Opening Files and Folders

https://superuser.com/questions/38984/linux-equivalent-command-for-open-command-on-mac-windows

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

Command Line Utilities are Awesome!

Blockchain price action SHA-256 satoshis public key freefall together few understand this transaction address. Double-spend problem hash, inputs pizza, hash difficulty sats Merkle Tree bitcoin? Difficulty mining space citadel, bitcoin deflationary monetary policy, roller coaster to the moon double-spend problem, address! When lambo, SHA-256, transaction.

Next Steps

Practice Makes Perfect!

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

Upgrade Your Terminal Colors and Prompt

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

Explore the Universe of Commands

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

Congratulations!

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

You Are Awesome!

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

Blockchain price action SHA-256 satoshis public key freefall together few understand this transaction address. Double-spend problem hash, inputs pizza, hash difficulty sats Merkle Tree bitcoin? Difficulty mining space citadel, bitcoin deflationary monetary policy, roller coaster to the moon double-spend

problem, address! When lambo, SHA-256, transaction.

Expanding the Commands Available to You

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

Installing Homebrew for Mac

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

Using Built-In Linux Package Managers

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

Installing wslu for Windows Subsystem for Linux

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

We can search the *distance* field in all of our star files to find stars that are greater than 500 light years from Earth by using a regular expression with the `grep` command. Notice in *Example 66* that the distance field has four numbers followed by the letters "ly", meaning light years. This example shows `0545ly`. The sequence of numbers represent light years expressed with four-digits—with thousands, hundreds, tens, and ones digits. So to search for numbers that are greater than 500, let's use a number range for each digit in our search pattern that we give to the `grep` command. The regular expression `[0-9][5-9][0-9][0-9]ly` is a little recipe that matches "a number from zero through nine in the thousands digit, followed by a number from five through nine in the hundreds digit, followed by a number from zero through nine in the tens digit, followed by a number from zero through nine in the ones digit, followed by the letters ly".

With this pattern defined, let's search all of the star files using an `*` (star) wildcard for file expansion. When you search multiple files, by default the `grep` command displays which file the lines are found in, and in this case we use the `-h` option to hide the file names from being shown in the results. Go ahead and give this a try! Have a look at *Example 67*, which shows the `grep` command syntax and the sorted results.

Example 69: Using the `grep` *command to search multiple files ending in* `txt` *using a regular expression (* `-E` *) pattern. The results are sorted numerically by the fourth field, distance in light years.*

```
chris@ophir signs % grep -h -E "[0-9][5-9][0-9][0-9]ly" *txt | \
sort -t "   " -k 4 -n
Xamididamura        16°30'-Sagittarius  blue-dwarf          0501ly  +3.00
Mirfak              02°26'-Gemini       yellow-super-giant  0506ly  +1.79
Albaldah            16°86'-Capricorn    yellow-bright-giant 0510ly  +2.88
Sadalmelik          03°41'-Pisces       yellow-super-giant  0524ly  +2.94
Sadalsuud           03°44'-Pisces       yellow-super-giant  0537ly  +2.89
Suhail              11°32'-Virgo        orange-super-giant  0545ly  +2.21
Antares             10°06'-Sagittarius  red-super-giant     0554ly  +0.91
Shaula              24°55'-Sagittarius  blue-sub-giant      0571ly  +1.63
Lesath              24°21'-Sagittarius  blue-sub-giant      0576ly  +2.65
Fang                03°17'-Sagittarius  blue-dwarf          0586ly  +2.91
Avior               23°28'-Virgo        orange-giant        0605ly  +1.86
Adid-Australis      06°01'-Gemini       blue-dwarf          0612ly  +2.89
Cih                 14°16'-Taurus       blue-sub-giant      0613ly  +2.15
Beta-Arae           24°33'-Sagittarius  orange-super-giant  0646ly  +2.80
Saiph               26°45'-Gemini       blue-super-giant    0647ly  +2.06
Enif                02°13'-Pisces       orange-super-giant  0690ly  +2.39
Mintaka             22°43'-Gemini       blue-white-giant    0692ly  +2.41
Alniyat             08°09'-Sagittarius  blue-dwarf          0697ly  +2.89
Alnitak             25°02'-Gemini       blue-super-giant    0736ly  +1.77
Menkhib             03°28'-Gemini       blue-super-giant    0752ly  +2.80
Aspidiske           11°44'-Leo          blue-super-giant    0766ly  +2.26
Ahadi               11°44'-Leo          orange-super-giant  0807ly  +2.70
Mebsuta             10°17'-Cancer       yellow-super-giant  0845ly  +2.98
Rigel               17°11'-Gemini       blue-super-giant    0860ly  +0.13
Wezen               23°45'-Cancer       yellow-super-giant  1600ly  +1.84
Sadr                25°10'-Aquarius     yellow-super-giant  1800ly  +2.23
Girtab              27°52'-Sagittarius  blue-giant          1930ly  +2.99
Deneb               05°39'-Pisces       white-super-giant   2600ly  +1.25
```

Stellar! The `grep` command grabbed every line from each file that matched the pattern, and we sorted the results to help us see that there are twenty-eight stars in the dataset that are 500 or more light years away—from Xamididamura at `501ly` to Deneb at `2600ly`.

To create this regular expression pattern, we used four number ranges followed by two alphabetical characters (`[0-9][5-9][0-9][0-9]ly`). The syntax for regular expressions is described in a manual page called `re_format` This manual page has all of the technical details, and we summarize

many of the common special character recipes and rules used to create a regular expression pattern in *Table 5*. Have a look at the table to get a sense of the format of these little tools that help you define patterns for searching text, and we will discuss the details.

Table 5. Commonly used regular expression symbol sequences used to match patterns.

Patterns	
abc…	All upper and lower letters and symbols
123…	All number digits from zero to nine
.	Any single character
[abc]	Only a, b, or c (all characters can be bracketed)
[^abc]	Not a, b, nor c (all characters can be bracketed)
[A-Za-z]	Characters A to Z or a to z (a range of letters)
[0-9]	Numbers zero to nine (a range of numbers)
p1|p2|p3	Any of pattern 1 OR pattern 2 OR pattern 3
(p)	A grouping of pattern p
Pattern shorthands	
\d	Any single number digit
\D	Any non-digit character
\t	A single tab character
\n	A single newline character
\r	A single carriage return character
\s	A single whitespace character (space, tab, newline, carriage return)
\S	A single non-whitespace character
Pattern modifiers	
p*	Zero or more repetitions of the preceding pattern p
p+	One or more repetitions of the preceding pattern p
p?	Zero or one of the preceding pattern p
p{m}	m number of repetitions of the preceding pattern p
p{m,n}	From m to n repetitions of the preceding pattern p

Let's familiarize ourselves with some of these little regular expression syntax gems, beginning with the first section of of *Table 5* labeled *Patterns*. The first two rows show that any individual letter, symbol, or number can be used in a regular expression pattern. These are referred to as *literal* characters. For instance, the pattern `-Leo` means "match a sequence of characters with a *dash* followed by an uppercase letter *L* followed by a lowercase letter *e* followed by a lowercase letter *o*".

Next, `[]` (square brackets) are used to create a list of matching characters where the order doesn't matter. For example, the pattern `[-Leo]` means "match any *dash*, uppercase *L*, lowercase *e*, or lowercase *o* characters, in any order". This takes a little practice, but remember that square brackets create a list of characters, not a sequence of characters. Also, when you start your square bracket list with a `^` (caret) symbol, it indicates you want to match any characters *not* in your list. For instance, the pattern `[^Tt]` means "match any character that is not an uppercase letter *T* or a lowercase letter *t*".

When creating patterns with literal characters, you may at times need to match the special characters used in the regular expression syntax. In these cases, use a `\` (backslash) character to "escape" the special meaning of the character. For instance, `\[` (backslash with left square bracket) will match a literal `[` (left square bracket) instead of using it to define a list or range of characters.

Next, using a `|` (pipe) symbol in between patterns lets you search for multiple patterns. For instance, the pattern `Leo|Aquarius` means "match the sequence of letters *Leo* OR the sequence of letters *Aquarius*". You can use as many pipe symbols as needed to search for multiple patterns.

Regular expressions used with commands like `grep` need to be wrapped in `""` (double-quotes) so they are passed as an argument to the command. This shields special characters like the `|` (pipe) symbol from being interpreted by the shell rather than being sent as an argument to the command.

Lastly, you can wrap a pattern in `()` parentheses in order to create a pattern group. This lets you treat whatever pattern is inside the parentheses as a single unit. We'll demonstrate this when we discuss pattern modifiers shortly.

Let's next discuss the pattern shorthands that are commonly used in regular expressions shown in the second section of *Table 5*. These are helpful, particularly when you need to match invisible characters. A `\d` will match a single number digit, and is the same as `[0-9]`. Likewise, `\D` will match any single non-number digit, which is equivalent to `[^0-9]`. Then, to cover each of the invisible characters possibly in text files, a `\t` matches a tab character, `\n` matches a newline character, and `\r` matches a

carriage return. These invisible characters are known as *whitespace* (along with a space character). You can use a `\s` as a catch-all to match any of these invisible characters. Then on the flip side, you can use `\S` to match any non-whitespace characters (meaning any visible characters).

As you can see, these little symbol combinations, when used together, let you build very powerful pattern matchers. The third section of *Table 5* shows some of the handy modifers you can use with your patterns which help you define optional or repeating patterns. Let's take the word `Sagittarius`. If we use a `*` (star) symbol in the pattern, it will match zero or more of the pattern that precedes it. For instance, let's use the pattern `it*`, which means "match a lowercase letter *i* followed by zero or more lowercase letter *t*":

```
echo "Sagittarius" | grep -E "it*"  # matches "itt" and the second "i"
Sagittarius
```

The `*` (star) means the `t` is optional, so this the pattern will match two places in the word `Sagittarius`, first the `itt`, and additionally the second `i` in the word:

If we replace the `*` (star) with a `+` (plus) sign, the letter `+t+` is required, so the pattern `+it` means "match the lowercase letter *i* followed by one or more lowercase letter *t*":

```
echo "Sagittarius" | grep -E "it+"  # matches the letters "itt"
Sagittarius
```

If we use parentheses to group our two letter pattern with a `(it)+`, this means "match one or more of a sequence with a lowecase letter *i* and a lowercase letter *t*":

```
echo "Sagittarius" | grep -E "(it)+"  # matches the letter sequence "it"
Sagittarius
```

If we want to indicate that a pattern is optional, we can use a `?` (question mark) symbol. The pattern `it?` means "match a lowercase letter *i* followed by zero or one lowecase letter *t*":

```
echo "Sagittarius" | grep -E "it?"  # matches the letters "it"
Sagittarius
```

As you can see, each of these sequences of symbols give us an extremely flexible set of tools for finding patterns in text. With practice, using these character symbol combinations becomes second-nature when searching for data.

Bitcoin ipsum dolor sit amet. Volatility bitcoin proof-of-work block reward few understand this public key sats cryptocurrency mempool. Mining digital signature full node nonce freefall together hyperbitcoinization whitepaper hash. Consensus blockchain hodl, when lambo timestamp server transaction deflationary monetary policy, bitcoin. Satoshi Nakamoto transaction, proof-of-work hard fork freefall together, block height stacking sats. Hyperbitcoinization decentralized price action?

Peer-to-peer, whitepaper block reward roller coaster hash, hodl Bitcoin Improvement Proposal, difficulty. Electronic cash wallet few understand this electronic cash roller coaster blocksize hash. Few understand this, block reward hyperbitcoinization, nonce block reward whitepaper cryptocurrency! Halvening roller coaster UTXO, difficulty when lambo UTXO genesis block. Halvening hyperbitcoinization Bitcoin Improvement Proposal Satoshi Nakamoto.

Proof-of-work block reward roller coaster wallet sats price action consensus mempool, hyperbitcoinization. To the moon volatility, sats outputs money printer go brrrrr freefall together electronic cash sats. Private key, double-spend problem, electronic cash public key, mining volatility timestamp server few understand this! Blockchain outputs halvening, proof-of-work hard fork satoshis roller coaster!

Soft fork mempool double-spend problem peer-to-peer bitcoin satoshis inputs hash hodl! Hashrate full node consensus blockchain public key halvening hyperbitcoinization bitcoin! Merkle Tree bitcoin, decentralized transaction halvening public key wallet hyperbitcoinization. When lambo satoshis hard fork, freefall together money printer go brrrrr.

Blockchain price action SHA-256 satoshis public key freefall together few understand this transaction address. Double-spend problem hash, inputs pizza, hash difficulty sats Merkle Tree bitcoin? Difficulty mining space citadel, bitcoin deflationary monetary policy, roller coaster to the moon double-spend problem, address! When lambo, SHA-256, transaction.