

The Path After Python

Where to go after learning python syntax



Eric Riddoch

This page is intentionally left blank.

Contents

About This Book

Who Is This Book For?

What Will Be Covered?

How to Use This Book

Part I - Linux and the Command Line

Chapter 0 - Getting Started With Linux

Introduction

What is Linux?

Why Linux? (Open Source vs Closed Source)

What is the Command Line?

Installing Linux

For Linux Users

For MacOS Users

For Windows 10 Users

Video Version

Text Version

1. Enable the Windows Linux Subsystem
2. Download Ubuntu Linux
3. "Virtualize" Ubuntu Using VirtualBox or VMWare

If You Have Windows 7 or Less

1. Dual Boot Linux
2. Use Git Bash

Validate Your Linux Terminal Setup

Chapter 1 - Navigating the Linux File System

Introduction

The bash Shell

The Linux File System

The pwd Command

1. Likely Output on MacOS
2. Likely Output in the WSL or Linux

What does this output mean?

ls and cd

Command 1: ls

Command 2: cd

More on File Paths

Absolute vs Relative File Paths

Special Folder Names

1. The Root Directory: /
2. The Home Directory: ~
3. The Current Directory: .
4. The Parent Directory: ..

Chapter 1 Wrap Up

Chapter 1 Exercise Solutions

- 1.1 - Exploring the File Tree

Chapter 2 - bash Command Arguments

Introduction

Placeholders <> and []

Comparing Commands to Python Functions

some_command

[FLAGS]

[KEYWORD ARGUMENTS]

[POSITIONAL ARGUMENTS]

Chapter 2 Wrap Up

Chapter 2 Exercise Solutions

- 2.1 - Creating the pictures/ Directory

Chapter 3 - Moving and Deleting Files

Introduction

- Introduction
- Moving Files With mv
- Renaming Files with mv
- Deleting Files With rm
- Story Time
- Chapter 3 Wrap Up
- Chapter 3 Exercise Solutions

- 3.1 - Creating the furniture directory
 - 3.2 - Renaming a file with a bad name
 - 3.3 - Aliasing rm
 - 3.4 - Practicing rm

Chapter 4 - Wildcards and Globbing

- Introduction
- Wildcards
- Wildcards in Context
- Wildcard 1 - * (asterisk)
- Wildcard 2 - ?
- Wildcard 3 - []
- Wildcards and Double Quotes
- Globbering in Python!
- Chapter 4 Wrap Up
- Chapter 4 Exercise Solutions

- 4.1 - Sorting Files by Type with Wildcards

Chapter 5 - The Power of Redirection

- Introduction
- A First Encounter with Redirection Operators
- Streams - The Communication Channels of bash Commands
 - Creating & Overwriting Files with >
 - Creating & Appending to Files with >>
- Taking Streams as bash Command Inputs
- Standard Input
- More Examples of Using Streams as an Input

The < Operator

The << Operator

The | (Pipe) Operator

Awesome Commands that Take Streams as Input

grep - The Ctrl+F of bash

pbcopy and pbpaste

pbcopy

pbpaste

nl

sort

wc

Chapter 5 Wrap Up

Chapter 5 Exercise Solutions

5.1 - Capturing the Output of ls with Redirection Operators

5.2 - "grepping" /bin

5.3 - Playing with nl, wc, and sort

About This Book

Welcome to this book about transitioning from a beginner python "coder" to a genuine (southern accent) python software engineer! Before we begin I want to say that this book is a lot less about *coding in python* than the title may have led you to believe.

Sure, there are some chapters on the syntax of classes and objects. However, the part of the "software learning curve" that beginners often struggle with the most are the *non-coding* aspects of software development. There is a **big** (but actually straightforward) jump you have to make to transition from writing small, buggy scripts on your laptop to writing giant code bases used by potentially thousands of people.

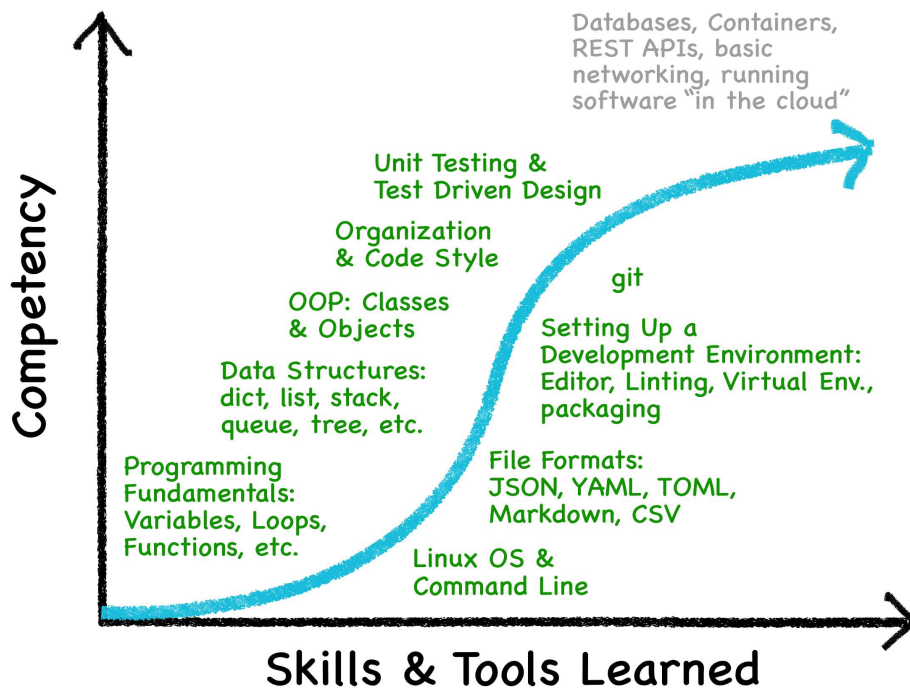
Who Is This Book For?

This book is meant to be a launch pad for "coders" who have learned the basic syntax of python, but don't necessarily know where to start when it comes to building large, reliable software to solve real world problems. For example, you may have learned python basics on your own or in a course, but you haven't studied "Computer Science" or "Software Engineering" in school. Many hobbyists, data analysts, data scientists, software interns, and even software engineers with a non-software background fall into this category.

What Will Be Covered?

So, what exactly is the difference between coders who struggle and those who feel like they can build anything? They say, "you don't know what you don't know." Here, I've tried to make the gap between experienced and inexperienced programmers more concrete by naming the key **technical skills** that I have seen make the largest difference.

The Software Engineer Learning Curve



How to Use This Book

It's written sequentially so as to build on itself chapter by chapter. You should be able to skip ahead if you already know the material of a previous chapter.

Part I - Linux and the Command Line

In this section, we will cover installing and setting up the *Linux command line*. Then, we will go over **bash** –the command language used to control your computer through the command line. Before moving on, please consider these two pieces of advice that I consider to be game-changing for my own tech career:

Tip

Takes digital notes with your own examples.

Alright, I'll say it. I love `bash`, but frankly, it is unintuitive and disorganized in many aspects.

There will probably never be a point that you can sit down and write a sweet, 30+ line `bash` script without Googling at least once. So, save your future self a good amount of time, and enhance your learning experience by writing down your exercise solutions, your interpretation of command syntax, difficult concepts, and anything that sticks out to you.

Keeping my own "personal knowledge base" in the form of digital notes has greatly increased my productivity when it comes to learning technical things and *applying what I learned* afterwards. As coders, we use Google a lot to find answers to particular questions. As you find these, record the links to the great videos, articles, and books, so that you can find them faster next time.

Also, videos are nice, but if you need to go back and find one specific piece of information buried in a 10-minute video, you might be stuck having to watch the whole thing over again. In addition to summarizing the useful takeaways in your notes, **take screenshots** of nice diagrams in both videos and books. Copy/paste to your notes liberally!

Screenshot Shortcuts

`⌘ Cmd` + `⇧ Shift` + `4` MacOS

`⌘ Cmd` + `⌘ Ctrl` + `⇧ Shift` + `4` MacOS (saves the screenshot right to your clipboard!)

`⊞ Win` + `⇧ Shift` + `S` Windows 10

Note Taking App Recommendations

Few note taking apps actually support syntax highlighting. I have personally used all three of these and loved them:

- **BoostNote** (free forever)
- **Quiver** (one time fee on MacOS)
- Pro version of **Notion** (recurring monthly subscription)

A cool thing about BoostNote is that you must take your notes in `Markdown`. We'll have a whole on that later in this book because of how useful it is. This book is actually written using `Markdown`, and I think it looks quite nice 😊

 **Tip****Do the exercises in your own terminal as you read.**

If you are new to the terminal or inexperienced with it, PLEASE follow along with the exercises, and resist the temptation to skip past them. While `bash` is powerful, it is notorious for having tricky nuances that lead to very confusing bugs.

By actually running the commands for yourself, you will develop your `bash` "muscle memory" and discover the aspects of `bash` that trip you up personally *before* you unknowingly cause a bug in a long script and have to spend hours debugging it.

With these out of the way, enjoy learning about Linux!

Chapter 0 - Getting Started With Linux

Introduction

Depending on your background, this chapter may supercharge your computer skills more than any other chapter in this book. The concepts we'll discuss here are the gateway to limitless power which will allow you to accomplish incredible computer-y feats with the greatest of ease. Here are *just a few* of such feats:

- run your own website
- host a Minecraft server that will be the envy of all your friends
- bulk download YouTube videos and music and convert them to other file formats
- understand answers to everyday tech problems on Google that baffle regular humans
- generate pretty PDFs like this one 😊
- download and run programs that are inaccessible without the command line
- organize, rename, move, and delete files on your computer incredibly quickly
- make your code available to the world to use
- **take advantage of tools that make writing code a joyful, fulfilling experience**

The last four bullet points make the ideas in this chapter essential to your development as a programmer (pun intended). I hope you're excited to get started! We have a lot to cover, but first you will need to know what **Linux** and the Linux **terminal** are.

What is Linux?

Linux is an operating system, or OS for short. Examples of operating systems are Windows (10, 8, 7, XP, Vista) and MacOS (Big Sur, High Sierra, El Capitan). An **operating system** is the software that enables a computer to use its hardware to run programs. Until you install an operating system on a computer, you cannot run any programs on it. Without an OS, a computer is not much more than expensive pile of metal.

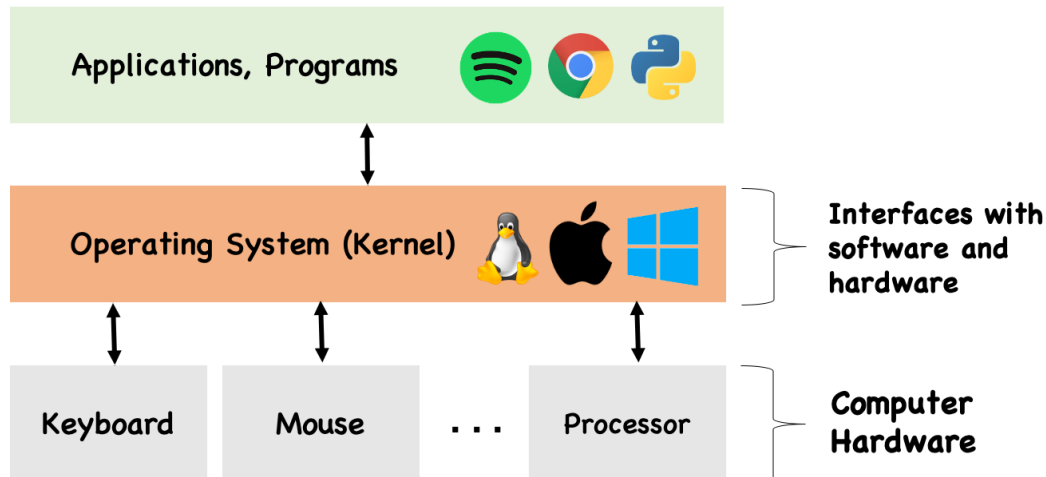
Note

It can be helpful to think of a computer as two components, *software* and *hardware*.

The hardware is the physical parts of your computer such as your keyboard, mouse, screen, hard drive, RAM, etc. The software is the programs that run on your computer, for example, Spotify, Microsoft Word, Google Chrome, the operating system, etc.

Here is a diagram showing the relationship between an OS, computer hardware, and software. See the penguin icon 🐧? That's Linux!

Operating Systems



Why Linux? (Open Source vs Closed Source)

You might wonder what the purpose of this third operating system is when most of the world seems to do just fine using Windows and MacOS. There are *many* fantastic reasons to use Linux for coding that all stem from the fact that Linux is *open source*.

Back in the early 90s, software developers were dissatisfied with proprietary ("closed source") operating systems for a number of reasons, and their response to this was Linux!

Here is some background on why an open source OS is desirable. If a closed source operating system is not stable and needs some kind of bug fix, you are stuck waiting until the company in charge of it decides to release a patch (if they ever do). Also, proprietary operating systems usually are not free (Windows) or they can only run on special hardware (MacOS).

In contrast, open source software is free, and if you find a bug that bothers you or a feature you wish it had, you can just write it yourself! Then, if the administrators of that open source project like your code, they can approve it so your new feature is available for everyone. There are various pros and cons to open source vs. closed source software, but in general, software developers love open source tools because they are free and have a whole community helping to make them better.

You can probably see why companies prefer to keep their software proprietary. If Microsoft opened up Microsoft Word to the world for all programmers to make it better... they would also have made it freely available, and so they would not make any money on it. Trade offs.

Today, Linux has blossomed into the most popular OS to run software on *behind the scenes*. That may strike you as strange, because it is highly possible you have never seen a computer running Linux in your life! That is because Linux is not as user friendly to non-coders, so MacOS and Windows prevail on most household computers. However, I can guarantee you that almost every website, mobile app, and internet-enabled program that you have ever used works because there is a computer running Linux somewhere far away powering it.

If you have never used Linux before, welcome to the club! Get ready to gain some computer superpowers and some serious "street cred" in the tech community.

What is the Command Line?

You've seen it before. Picture an intense movie scene. There's critical information the coder can extract from the disk, but only if he can decrypt it! He furiously types 1's and 0's onto an intimidating black screen. Suddenly, the disk clicks and confetti bursts out. He's done it! Everyone cheers 🎉🎉🎉🎉



Definitely what coding looks like...

Drum roll please! Introducing. The. Command line!!! Also known as the terminal.

That's right. I know it doesn't look like much, but this black and green window is the key to unlocking the computer-y superpowers I've been promising you. Unlike Windows and MacOS, you aren't meant to control Linux by clicking around on pictures of folders and dragging things this way and that way. Instead, we will control Linux by typing commands into the terminal, *and that's a good thing!*

So, with that said, let's get your terminal set up!

Installing Linux

Before we can get into the fun part of using Linux, we will need to install it. The instructions differ depending on the OS you have on your computer. Read below and follow the instructions matching the OS you are currently running.

Tip

Sadly, there is no way I can include solutions to every possible error you might see during the install process.

However, what we are attempting to do here is extremely common. So, if something does happen, take heart and copy/paste as much of the error message as you can into ye ol' Google. There should be a host of YouTube videos and StackOverflow answers leading you on to success.

For Linux Users

Why are you even reading this chapter?

For MacOS Users

Actually, MacOS is Apple's proprietary flavor of Linux! That may not make sense, so don't worry about it right now. We'll cover *Linux distributions* in a later section. Just know that you have it a lot easier than Windows users at this stage.

Use `⌘ Cmd` + `Space` to open up **Spotlight Search** and then search for **terminal**. And you're done!

For Mac Users

If you want a program that is slightly nicer than **Terminal**, go to <https://iterm2.com/> and download **iTerm2**.

If you choose to use **iTerm2**, you should open up **iTerm2** any time we say "the terminal" or "the command line" moving forward.

For Windows 10 Users

Before Windows 10, it was awkward to use Linux on a Windows computer (See the [Windows 7 Section](#) for details). However, due to the high demand from developers, Microsoft came up with a solution that allows you to run Windows and Linux at the same time. They call this the **Windows Linux Subsystem** or **WSL** for short.

Warning

Installation processes change somewhat frequently. If you find that these installation instructions are no longer up to date at the time you read this, Google search "How to install the Windows Linux Subsystem".

Video Version

When it comes to clicking on various things to install software, I personally prefer to watch a video. This ~3.5 minute [YouTube video](#) does a great job demonstrating the installation.

Text Version

You can find the official Microsoft version of these instructions [here](#).

1. ENABLE THE WINDOWS LINUX SUBSYSTEM

1. Log onto your beautiful computer running Windows 10.
2. Search for `Powershell` in your search bar.
3. Right click on `Powershell` and click `Run as Administrator`. Note, at this point, you may be prompted to use your administrator password.
4. Copy this command into `Powershell` and hit `Enter ↵`. Then, type `Y` when it asks you if you are sure and hit `Enter ↵` again.

```
dism.exe /online /enable-feature /featurename:Microsoft-Windows-Subsystem-Linux /all
```

5. If your computer prompts you to restart, do it.

2. DOWNLOAD UBUNTU LINUX

1. Launch the `Microsoft Store`
2. Search `Ubuntu`
3. Click `Ubuntu 18.04` and click `Get`
4. Now launch it! (Find `Ubuntu` in the task bar to launch it)
5. **Important!** Ubuntu will prompt you to set a username and password. I highly recommend using the same username and password that you use on your Windows OS user so that it is easier to keep them straight.

3. "VIRTUALIZE" UBUNTU USING VIRTUALBOX OR VMWARE

`VirtualBox` and `VMWare` are both programs that let you run other operating systems at the same time as your "host" operating system—which, in your case, is Windows 7 or less. Technically this is the same technology that the Windows Linux Subsystem uses on Windows 10.

However, the WSL is much more tightly integrated with Windows than Linux would be if you used one of these two programs. Due to this, you would have an *okay* experience learning and using Linux for development, but it would be a pain compared to using the WSL.

There are a multitude of tutorials for these two tools online.

If You Have Windows 7 or Less

Sadly, if you are running Windows 7 or less, your options are quite limited. Microsoft did not add the "Windows Linux Subsystem" until Windows 10. And since, unlike MacOS, Windows is not a special version of Linux, you cannot run Linux commands directly on Windows and expect the same results.

Here are two options:

1. Dual Boot Linux

Download Ubuntu Linux onto a USB drive and "dual boot" it. This means that whenever you turn your computer on, you will have a choice between starting up in Windows and starting up in Ubuntu Linux. Beware that there is a risk of corrupting your hard drive if you do this wrong. There are many tutorials online and on YouTube if you want to go this route.


This book covers the fundamentals of Linux from the software developer perspective, but if you dual boot Linux, you will want to know much more. If you are committed, you have a long Linux road ahead of you, but you will come out knowing it better than everyone else.

2. Use Git Bash

Head over to <https://git-scm.com/download/win> and install "Git for Windows". This is a very useful program that we will actually be using later in this book. When you install it, it will also install a program called "GitBash". This is a program that *simulates* a Linux command line on Windows.

Unfortunately, Git Bash is not a true Linux command line, so many of the Linux commands will not work in it.

Validate Your Linux Terminal Setup

Once you have any of these solutions working to use your Linux terminal, type in the following command and hit 

```
echo "Hi, terminal\!"
```

If the text `Hi, terminal!` prints out on the line below, congratulations!! You have a working terminal! In the next chapter, we will give some more background on the terminal and start talking about how to use it.

Chapter 1 - Navigating the Linux File System

Introduction

In this chapter, we will introduce the `bash` shell and use `bash` commands to navigate around in Linux using the terminal.

The `bash` Shell

So far, we have talked about Linux as an operating system, and we have established that the main way users are meant to control Linux is through the command line. The only command we have seen is `echo` which prints a message to the console (another name for the command line). As it turns out, the `echo` command belongs to a scripting language called `bash`. `bash` commands are the ones we will be learning in part one of this book.

As it turns out, there are several command languages—or `shells`—that can be used to control Linux. After `bash`, the next most popular shells are `zsh` (z-shell) and `fish`. It is safe to say that `bash` is the industry standard when it comes to writing scripts for Linux.

The Linux File System

I am sure you have noticed that Windows and MacOS organize files using folders. Linux does the same. We call the complete set of files and folders on an operating system a `file system`.

Note

`Directory` is a synonym for "folder" when talking about a file system.

Here is an example file tree with a few folders.

Name	Date Modified	Size	Kind
▼ unknowns	Today at 4:55 PM	--	Folder
▶ string cheese	Today at 4:53 PM	--	Folder
▶ tomatoes	Today at 4:53 PM	--	Folder
▶ fruits	Today at 9:12 PM	--	Folder
▼ vegetables	Today at 9:12 PM	--	Folder
▶ carrots	Today at 4:52 PM	--	Folder
▶ celery	Today at 4:52 PM	--	Folder
▶ asparagus	Today at 4:52 PM	--	Folder
vegetable summary.txt	Today at 5:06 PM	Zero bytes	Plain Text
summary.txt	Today at 5:06 PM	Zero bytes	Plain Text

A File Tree

We can also represent a file tree in text like this:

```

pictures
├── fruits/
├── unknowns
│   ├── string\ cheese/
│   └── tomatoes/
└── vegetables
    ├── asparagus/
    ├── carrots/
    ├── celery/
    ├── summary.txt
    └── vegetable\ summary.txt

```

We will refer back to this example file tree several times in the next sections.

Note

Note that including a `/` at the end of a folder name is optional. If a file name ends in a `/`, you know it's actually not a file, but a folder. In this text diagram, we would not be able to tell whether `fruits/`, `tomatoes/` and `string\ cheese/`, etc. are folders without the `/` at the end.

The `pwd` Command

It is time for our second `bash` command! Open up your terminal and type `pwd`. This is short for "print working directory".

1. Likely Output on MacOS

```

$ pwd
/Users/eric

```

2. Likely Output in the WSL or Linux

```
$ pwd
/home/eric
```

Note

The `$` in front of `pwd` is not part of the command. It is supposed to inform you that you are in a `bash` terminal, and therefore able to enter `bash` commands.

Danger

In your terminal, if you see a `#` instead of a `$`, then you are running your terminal as the "root" (or admin) user which is dangerous. The root user has permission to delete or change any file on your system, which could do serious harm if you edited the wrong files.

Type `whoami` to get the user you are acting as in your terminal session.

What does this output mean?

It is okay if your output from `pwd` does not match the examples above. Whatever you got, this output is the folder you are currently inside of in your terminal AKA your **working directory**. We call this sort of string a **file path**, which is a path to the location of a file or folder on your computer.

The file path format is pretty self-explanatory: it is a sequence of nested folders separated by `/`s. The very last name in the sequence can be a file or a folder.

If folder **A** contains a file or folder **B**, we call **A** a **parent directory**. In this case, we would also call **B** a **child directory** or **child file**, because it is contained in **A**.

Here is another file path example using the file tree from earlier. We are assuming that `pictures/` is inside of `eric/`:

```
# the file path to summary.txt
/Users/eric/pictures/vegetables/summary.txt
```

We will see more examples in the next section.

Note

The reason that, after running `pwd`, outputs similar to `/Users/eric` and `/home/eric` are "likely" is because the `bash` shell usually starts you out inside your user's *home directory*.

On the main operating systems—Windows, Linux, and MacOS—each user on one computer has a folder named after them. All of a user's files and folders are created here by default. We call this your user's **home directory**. Only your user and the "root" user have the ability to create, edit, and delete files inside of your home directory.

ls and cd

`pwd` shows us which folder we are currently inside of in the terminal—we are going to call this our "working directory" or "current directory" from now on. We can "move around" inside the file system using the `ls` and `cd` commands.

Command 1: ls

`ls` is short for "list". It shows you all the files and folders in your working directory.

For example, if I were inside of `/Users/eric/pictures/vegetables`, I would get the following output from `ls`:

```
$ ls
asparagus carrots celery summary.txt vegetable\ summary.txt
```

There are often *hidden files and folders* in your home directory. **Hidden files and folders** have names starting with `.`. By default, hidden files and folders do not show up when you browse your file system both when you use your mouse to click on folders, and when you type `ls`.

To check for hidden files and folders in your current directory, use `ls -a`.

```
$ cd /Users/eric
$ ls -a
# you will likely see more than this in your user home
. .. .local pictures
```

We will talk about the `.` and `..` results in the [section on special folder names](#).

Command 2: cd

`cd` is short for "change directory". Knowing that I'm inside of `/Users/eric/pictures/vegetables`, I can move into the `celery/` folder like this:

```
$ cd celery
$ pwd
/Users/eric/pictures/vegetables/celery
```

Success! If I *hadn't* been inside of `/Users/eric/pictures/vegetables` already, I could have typed out the full path to `celery/` like so:

```
$ cd /Users/eric/pictures/vegetables/celery
$ pwd
/Users/eric/pictures/vegetables/celery
```

We will look closer at the difference between these two approaches in the [section on absolute and relative file paths](#).

More on File Paths

Before we look at more example file paths, we need to define two types of files paths (there are only two). For context, let's look at my home directory again.

On my machine, my home directory is at `/Users/eric`. This seems like an organized way to do things: there is clearly a folder somewhere on my machine called `Users`, and it makes sense that my user folder (`eric`) belongs in there.

But is the `Users` folder inside of anything else? If it is, is the `Users` folder's *parent* directory have a parent directory? How far up the file tree can we go?

The answer to all these questions is `/`. Yes, I know, very descriptive. But seriously, `/` is actually a folder called the `root directory`. There is nothing containing `/`. On Linux-style operating systems, `/` is where the entire file tree begins.

Note

Did you catch that the file system is shaped like a tree 🌲, and the folder containing all folders is called the "root"? Deep 😊

Now that you know what `/` means, we can finally define the two types of file paths.

Absolute vs Relative File Paths

I mentioned earlier that there are two types of files paths. They are:

1. **Absolute File Path**: A file path starting with `/`. If you see that a file path starts with `/`, you know *exactly* where it is on your system.
2. **Relative File Path**: Relative file paths do not start with `/`. Let's elaborate on this.

Because relative file paths do not start with `/`, they are "relative" to your working directory (the output of `pwd`).

For example, assume that the `pictures/` directory from earlier is inside of `eric/`. We would like to know the path to `summary.txt`. If `pwd` shows you that your working directory is `/Users/eric`, then instead of writing out the full absolute path

```
# absolute path to summary.txt
/Users/eric/pictures/vegetables/summary.txt
```

we can actually just write

```
# path to summary.txt relative to /Users/eric
pictures/vegetables/summary.txt
```

⚡ We Hate Spaces In Paths 🚫

Programmers *hate* spaces in names and file paths.

If you want to turn a programmer against you for life, go put spaces in all of his/her file and folder names, and also in any database column names and dictionary keys while you are at it.

Spaces in names were fine back when you were a regular human, but now that you are becoming a software engineer, you must follow a more enlightened path of hate.

Wait, what?

Let's say your fingers betray you and you name a folder "string cheese". Now suppose you try to `cd` into it:

```
$ cd string cheese
cd: no such file or directory: string
```

Yep! That pretty much sums up the problem. It looks like we are trying to pass two separate arguments to `cd`: `string` and `cheese`... because spaces are the only way `bash` knows how to tell arguments apart. And then here you come putting spaces in your folder names and confusing the poor `bash` interpreter. Not cool.

To get around this issue, you will need to either rename `string cheese` to something else or put quotes around it:

```
# friends don't make friends have to type quotes when using cd
$ cd "string cheese"
```

Try to form the habit of using names like `string-cheese`, `string_cheese`, `stringCheese`, etc. I personally think hyphens are the prettiest, but technically the underscores are more "pythonic". And the `camelCase` style? Pssh! What is this? Javascript? 😏

Special Folder Names

There are four special folder names that have a particular meaning:

1. The Root Directory: `/`

`/` The root directory. `cd /` will always take you to the root directory.

2. The Home Directory: `~`

The home directory. `~` is an *alias* of home directory belonging to the user currently using the terminal. An **alias** is a nickname or shorter version of something.

If you ever get lost, `cd ~` (or even just `cd`) will always take you home.

Here is yet another relative path to the vegetable `summary.txt` file:

```
~/pictures/vegetables/summary.txt
```

3. The Current Directory: `.`

No matter where you are... `.` is where you are. Observe:

```
$ pwd
/Users/eric

$ cd .
$ pwd
/Users/eric

# we're going into this folder so many times!!
$ cd ../../../../../../
$ pwd
/Users/eric

$ cd ./pictures
$ pwd
/Users/eric/pictures

# stylish equivalent of "cd pictures/fruits" 😊
$ cd /Users/eric
$ cd ./pictures/../fruits
$ pwd
/Users/eric/pictures/fruits
```

4. The Parent Directory: `..`

No matter where you are, `..` means "parent directory" or "up a directory". For example:

```
$ cd ~
$ pwd
/Users/eric

$ ls
pictures
```



```
$ cd ..
$ pwd
/Users

# pointlessly convoluted way of getting to celery
$ cd eric/pictures/unknowns/../vegetables/celery
$ pwd
/Users/eric/pictures/vegetables/celery

# go up two directories (to pictures/) and then into fruits/
$ cd ../../fruits
$ pwd
/Users/eric/pictures/fruits
```

Press `Tab →` All the Time For Autocompletion!

Any time you are typing out a long path, press the `Tab →` key! And then press it again and again! In the terminal, `Tab →` **autocompletes commands for you**. It works especially well with paths to files and folders.

Exercise 1.1

Use `cd` and `ls` to explore your file system for a while. Here are some things to look at:

1. What are all the folders in the root directory (`/`)? Suggestion: you might find `/bin` interesting.
2. Explore your own folders that you normally go through using a mouse.
3. Play with the **4 special paths**.
4. Become addicted to the `Tab →` key! There should be a dopamine spike in your brain everytime you touch that key 😊

For Windows Linux Subsystem Users

The full explanation here is complicated (and very cool), but suffice it to say that your home directory inside the WSL is not the same as your Windows home directory where you keep your `Desktop`, `Documents`, and other main folders.

The Windows 10 `C:\` drive is "mounted" to the WSL file system at `/mnt/c`. This means that your Windows home directory should be at `/mnt/c/Users/<your username>`. So mine would be at `/mnt/c/Users/eric`. Likewise, your windows equivalent of `/` is at `/mnt/c/`.

Chapter 1 Wrap Up

Great job making it this far! In this chapter, we saw how to move through files and folders on a Linux system using `cd` and `ls`. The ability to navigate the Linux file system is foundational

to being able to take advantage of the cool parts of Linux.

In next chapter, we will dig into the format of `bash` commands so you can learn them very quickly.

Chapter 1 Exercise Solutions

1.1 - Exploring the File Tree

Nothing to see here. The exercise was just to explore your file system using the terminal.

Chapter 2 - **bash** Command Arguments

Introduction

We have officially seen five **bash** commands so far: **echo**, **whoami**, **pwd**, **cd**, and **ls**. Generally they have all followed the same pattern: you type the command first, and sometimes you type some extra arguments after the command to be more specific about what you are trying to do.

For example, **echo "hi"** has the string **"hi"** as an argument. **ls -a** has a weird **-a** argument that causes it to print out hidden files in addition to the usual output of **ls**. In this chapter, we will discuss what the different types of **bash** command arguments are, and how to use them.

Once you understand the command argument types, you will be able to glance at short summaries of what a command does and immediately understand how to use it. Let's get into it!

Placeholders **<>** and **[]**

When you Google for help with certain **bash** commands, or when you type a command incorrectly, you will often see abbreviated explanations like this:

```
# takes you to a path of your choice
cd [PATH]
```

or

```
# takes you to a path of your choice
cd <PATH>
```

This is a general format that tells you what you are meant to plug in as an argument for the **cd** command. A lot of **bash** beginners get confused by this format and end up typing something like:

```
$ cd [some/awesome/path]
# fail...
```

or

```
$ cd <some/awesome/path>
# fail...
```

The brackets (`[]`) and angle brackets (`<>`) are just placeholders to make the general command format look cleaner. Using this placeholder format, you can leverage what you already know about Linux (such as what a file path is) to quickly grasp what a new command does.

⚠ Exercise 2.1

Now that you have seen some example `bash` commands and know about how, here are two commands to use right now: `touch` and `mkdir`.

Their usage is like this:

```
# creates files at one or more paths
touch <file path 1> <file path 2> ...
```

```
# create directories at one or more paths
mkdir <directory path 1> <directory path 2> ...
```

Weren't those explanations concise? That should be enough information to know how these work now that you know about paths and basic command formats.

Task

Use these commands to create the `pictures/` directory and all of the files and folders. While you do this, play with using both relative and absolute paths. When you are finished, you should have a directory structure like this:

```
pictures
├── fruits/
├── unknowns
│   ├── string\ cheese/
│   └── tomatoes/
└── vegetables
    ├── asparagus/
    ├── carrots/
    ├── celery/
    ├── summary.txt
    └── vegetable\ summary.txt
```

If you get stuck, don't feel bad about looking at the [solution at the end of the chapter](#). Just know that you will feel like more of a champion 🏆 (and you will learn much more) if you try it on your own first.

Are You Using `Tab →` ?

The `Tab →` key's autocompletion works when you use `touch` and `mkdir`. When you are typing out the part of the file or folder path that already exists, `Tab →` can speed you through it.

Comparing Commands to Python Functions

You can think of `bash` commands like individual Python functions. Sometimes you can even think of them as a whole module of related functions. There is a mantra of good software design that says: every program, function, variable, class should **"do one thing, and do it well."**

The original implementers of the standard set of `bash` commands were very intentional about following this idea. Each command covers a set of functionalities that are very closely related, and nothing outside of that scope.

For example, the `ls` command can list non-hidden files and hidden files; it can list the attributes of files such as their size, date created, and access permissions; it can sort files by size, or recursively list all of the files and folders in subdirectories; and it can do *many* more highly niche operations that all have to do with listing.

The point here is that `ls` only does listing, and it is awesome at it! It is a one stop shop for all your file/folder listing needs. So how do we do all these things with `ls` ?

You may have guessed that the way we access all of these functionalities is by passing very specific arguments to `ls`. To see all of the arguments and functionalities of a `bash` command, you can use the `man` command as shown below. `man` is short for `manual`. You can use it to look up all of the "General Commands" that come with `bash`.

Tip

You can press `Q` in your terminal to exit out of the `man` output.

```
$ man ls
```

Here is some abridged output. You only need to get the picture of what a "`man` page" looks like and how you might use it yourself.

```
NAME
  ls -- list directory contents

SYNOPSIS
  ls [-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz1%] [file ...]

DESCRIPTION
```

```

...
The following options are available:
...
-a      Include directory entries whose names begin with a dot (.).
-p      Write a slash ( '/') after each filename if that file is a
directory.
-R      Recursively list subdirectories encountered.
-S      Sort files by size
-l      (The lowercase letter `ell'.) List in long format. (See
below.) A total
        sum for all the file sizes is output on a line before the long
        listing.
... much more

```

Did you see the `[-ABCFGHLOPRSTUW@abcdefghijklmnopqrstuvwxyz1%]` part of the man page?? Those represent all of the options available for the `ls` command. That is a ridiculous amount of options! (not in a bad way). It looks like they almost had to use every letter of both the upper- and lowercase alphabets to cover them all. Do you see `a` in there? We used that one!

Since we've already talked about placeholders, you might already sense that the `[]` brackets are unnecessary.

After seeing this list, you may be wondering: "there certainly are a lot of options here, are we able to multiple options at once? What would that look like?" Yes! Let's breakdown all types of `bash` arguments now.

Like python functions, `bash` commands can take both conditional and keyword arguments. Just like with python functions, these arguments can have multiple types: strings, integers, booleans, etc. Technically, anything you type into the terminal goes into a command as a string, but the command is able to parse that string to the type that it needs as long as your string is formatted correctly.

Your typical `bash` command format looks something like this:

```
some_command [FLAGS] [KEYWORD ARGUMENTS] [POSITIONAL ARGUMENTS]
```

We will step through this from left to right.

`some_command`

Self-explanatory. Examples are `echo`, `ls`, `cd`, etc.

[FLAGS]

The term "flag" is how `bash` scripters refer to *boolean arguments*. If `ls` were a python function, it might look something like this:

```
def ls(a=False, l=False, ... other arguments):
    if a:
        # show all the hidden files
    if l:
```

```
# show the list of files in "long mode"
...
```

In python, if we wanted to list all the directories with `ls()`, we would pass `True` in for `a` like so:

```
# list all the directories
ls(a=True)
```

In `bash`, specifically for boolean arguments, we do not type some equivalent of `True` or `False`, to `ls`. If we include `-a` in our command, then we are setting `a` to `True`. If we don't, then `a` defaults to `False`.

If we want to use multiple flags we can do it in two ways:

```
# set both the "l" and "a" flags to True
$ ls -l -a

# same thing
$ ls -la
```

Pretty easy right? Note that, as with Python, the order of these flags does not matter. `ls -la` is the same as `ls -al` is the same as `ls -a -l`.

Note

Often, flags have a longer alias that is more descriptive than just one letter. For example, `-v` is usually short for `--verbose`. This usually tells a `bash` command to print out extra information as it runs so you can get a clear picture of everything the command is doing.

98% of the time, the longer version is written with two dashes `--` rather than just one. Some commands do not follow this single vs double dash pattern, and that is frustrating for us regular people who like patterns because they make things easier to memorize 😞

[KEYWORD ARGUMENTS]

Also, like in python, a lot of `bash` commands are able to accept keyword arguments. `ls` has no keyword arguments, so for now I will make up a new command to use as an example.

Suppose we had a command called `download-youtube-video`. You can probably guess what it should do: download youtube videos.

Just like with Python functions, some keyword arguments may be required while others have default values. Also, for most commands, it does not matter which order you write the `[OPTIONS]` in.

We will say that the "`man` page" for the `download-youtube-video` command looks like this:


```

$ man download-youtube-video

NAME
    download-youtube-video - download videos from youtube.com

SYNOPSIS
    download-youtube-video [OPTIONS] <VIDEO_URL>

FLAG OPTIONS
    -b, --backwards
        Downloads the video... but it plays backwards!
    -a, --audio-only
        Only downloads the audio
    -v, --verbose
        Show a progress bar as the video downloads

KEYWORD OPTIONS
    -s, --speed <speed>
        The video playback speed: 1.0x / 1.25x / 2.0x
        Defaults to 1.0x
    -q, --quality <quality>
        Video quality: 140p / 540p / 720p / 1080p / 4k
        Defaults to highest available
    -o, --output-file-name <file path>
        (REQUIRED) Path name of the downloaded video/audio file
    ...

```

This would be an awesome command, right? If I ever use a command line tool that you create, and the arguments it takes are formatted this well, I will hug you. The arguments are so easy to understand!

But just in case they are not, below are some examples of how we would use this command.

Note

It almost *never* matters which order you write the optional **[FLAGS]** and **[KEYWORD ARGUMENTS]** in. In this case, **[OPTIONS]** is a lazy way to refer to both of those.

It also never matters whether you use the long form **--audio-only** or the short form **-a**.

Use any or all of these. Chances are, as you use the command line more, you will get lazy and only use short forms only. But you are your own person and you can make that sort of weighty decision yourself 😊

Try to download a dancing penguin video 🐧

```

$ download-youtube-video https://youtube.com/video/dancing-penguin
Error! Missing required keyword argument --output-file-name

```

Download the dancing penguin video, but make sure the video goes to the right place with the right name. **Note** that we can use back slashes `\` to spread a command across multiple

lines.

```
$ download-youtube-video \  
  --output-file-name /Users/eric/videos/dancing-penguin.mp4 \  
  https://youtube.com/video/dancing-penguin \  
SUCCESS!
```

Download the dancing penguin video backwards at 4k.

```
$ download-youtube-video \  
  -b --output-file-name /Users/eric/videos/backwards-dancing-penguin-HD.mp4 \  
  https://youtube.com/video/dancing-penguin \  
SUCCESS!
```

Only download the sweet soundtrack of the dancing penguin video, but backwards.

```
$ download-youtube-video \  
  --backwards --audio-only \  
  --output-file-name /Users/eric/videos/backwards-dancing-penguin-HD.mp3 \  
  https://youtube.com/video/dancing-penguin \  
SUCCESS!
```

[POSITIONAL ARGUMENTS]

The last argument type is positional arguments. **Positional arguments** are arguments that do not have a key-label like `--url-of-video-to-download <url>` to identify which parameter the argument is being passed in for.

In our `download-youtube-video` command, `VIDEO_URL` is a positional argument. Since it has no key-label, the only way the `download-youtube-video` command knows that `https://youtube.com/video/dancing-penguin` is meant as the `VIDEO_URL` is because it is the *first* unnamed argument.

With many commands, you can actually put keyword arguments both before and after the positional arguments like so:

```
$ download-youtube-video \  
  --backwards \  
  https://youtube.com/video/dancing-penguin \  
  -o backwards-dancing-penguin.mp4
```

This argument ordering can be confusing to read, but it is often valid. You should also know that many commands are picky about which order the positional and non-positional arguments go in. So, the only way to know for sure if your ordering of arguments will work is to read the `man` page or just try running it.

Chapter 2 Wrap Up

Great work reading through this chapter! Knowing how the different argument types work is a superpower that will help you learn new commands with rocket-like efficiency 🚀.

Chapter 2 Exercise Solutions

2.1 - Creating the pictures/ Directory

The directory looks like this:

```
pictures
├── fruits/
├── unknowns
│   ├── string\ cheese/
│   └── tomatoes/
└── vegetables
    ├── asparagus/
    ├── carrots/
    ├── celery/
    ├── summary.txt
    └── vegetable\ summary.txt
```

Here is the solution with commentary. The `Tab →` key sure is nice here 😊

```
# go to your home folder
$ cd ~
# make the pictures directory using a relative path from ~
$ mkdir pictures
# go inside of pictures
$ cd pictures
# create the next directories using paths relative to ~/pictures
$ mkdir fruits
$ mkdir vegetables
$ mkdir unknowns
# populate the unknowns/ folder using absolute paths for fun
$ mkdir /Users/<your username>/pictures/vegetables/tomatoes
# note that this path must be in quotes because of the space in "string
cheese"
# another option is to put a "\" before the space to escape it as in python,
# but that can get difficult to keep track of in longer paths. Without the
# space, bash would think "/Users/<your username>/pictures/vegetables/string"
# and "cheese" are two separate arguments.
$ mkdir "/Users/<your username>/pictures/vegetables/string cheese"
# take advantage of the fact that mkdir can accept multiple arguments
$ cd vegetables
$ mkdir asparagus carrots celery
# if "vegetable summary.txt" were not in quotes, touch would create
# a file called "vegetable" and a file called "summary.txt" because of the
space.
$ touch "vegetable summary.txt"
$ touch summary.txt
# DONE!
```

 **Tip**

Try using `ls -R ~/pictures` to recursively list everything in your `~/pictures/` to look over your work 😊

Chapter 3 - Moving and Deleting Files

Introduction

Earlier in the book, I told you that `bash` is used to control your computer entirely from the command line. That means that you should be able to do everything from the command line that you can do by clicking around with a mouse. Let's check how we are doing with that.

Usually, with a mouse, you do things like the list below. The ✓'s are things we have covered, the ✗'s are topics that we will get to, and the 😊's are topics we'll get to that you can only do in the terminal.

- ✓ Navigate through files and folders
- ✓ Create new files and folders
- ✗ Move, delete, and rename files and folders
- ✗ Run programs
- 😊 Automate the above things with scripts in mind blowing ways

In this chapter, we'll tackle moving, deleting, and renaming files. After that, we will almost be caught up with what you can do without the terminal.

Introduction

So far, the terminal may not seem that impressive to you. I mean, sure you feel like a hacker when you use it, but so far all we have done is navigate around and create files. You have probably been happily doing that without the terminal for years.

In this chapter, we will look at one of *many* things that the terminal is just better at: moving and deleting files **in bulk**. We will also cover a `bash` concept called `wildcards` and have a lively discussion about quotation marks. Let's get into it!

Moving Files With `mv`

To move files, we use the move command: `mv`. Its usage is like this:

```
mv <source path> <target path>
```

For example, with a folder structure like this:

```
furniture
├─ couch.jpeg
├─ bean-bag.jpeg
├─ chair.jpeg
├─ seats/
└─ storage/
```

Here are three equivalent ways of moving `chair.jpeg` into `seats` (assuming `furniture/` is our working directory).

```
$ mv chair.jpeg seats
$ mv chair.jpeg seats/
$ mv chair.jpeg seats/chair.jpeg
```

Result:

```
furniture
├─ bean-bag.jpeg
├─ couch.jpeg
├─ seats
│   └─ chair.jpeg
└─ storage
```

Actually, the usage for `mv` is more flexible than that. We can move multiple files at once when we use it like this:

```
mv <source path 1> <source path 2> ... <target path>
```

For example:

```
$ mv couch.jpeg bean-bag.jpeg chair.jpeg seats
```

Result:

```
furniture
├─ seats
│   ├── bean-bag.jpeg
│   ├── chair.jpeg
│   └─ couch.jpeg
└─ storage
```

Not bad! Imagine how awful it would have been to type the full `mv` three separate times 🙄

Note

We did not show an example of this, but you can also use `mv` to move folders inside of each other with the exact same usage.

Exercise 3.1

Create the directory structure from this example:

```
furniture
├── seats
│   ├── bean bag.jpeg # have a space in the name this time
│   ├── chair.jpeg
│   └── couch.jpeg
└── storage
```

BUT! Start from these commands to create them in a "flat" folder structure first. **Flat** means not nested. Use the `mv` command to put the files and folders in the right places.

```
$ mkdir furniture seats storage
$ touch "bean bag.txt" chair.jpeg couch.jpeg
```

[See the solution](#)

Renaming Files with `mv`

The `mv` command is also the way you rename files and folders with `bash`. You can think of renaming a file as moving that file from its original name to a new name (because names are really paths in Linux... deep).

For example, consider this file tree.

```
furniture
├── couch.jpeg
├── bean-bag.jpeg
├── chair.jpeg
└── seats/
```

Here is what renaming looks like:

```
# rename couch.jpeg to stool.jpeg
mv couch.jpeg stool.jpeg

# move chair into seats/ and rename AT THE SAME TIME!!!
mv chair.jpeg seats/futon.jpeg
```

The result will be this:


```
furniture
├── bean-bag.jpeg
├── stool.jpeg
├── seats
│   └── futon.jpeg
```


If this all makes sense, then you are ready to learn to delete files. Be careful!

Exercise 3.2

Create a file called `bad filename.txt`. Now rethink your life and get rid of the spaces by renaming it to `bad-filename.txt`.

[See the solution](#)

Deleting Files With `rm`

Just as the field of magic has a powerful yet dangerous subfield of dark magic, `bash` has a simple yet dangerous way of deleting files permanently: the `rm` command .

Danger

If you want to **destroy your entire system**, go ahead and run this command:

```
sudo rm -rf /
```

Continue reading to learn what this does.

Before we talk about the usage of `rm` you should know what we mean when we say "delete". When you "delete" files and folders on Windows and MacOS, the files go to a happy place called the "recycle bin" or the "trash can". This is essentially a waiting place that files go to before they make their journey to the great beyond.

On Windows and MacOS, when you *truly* want a file off of your computer, you click "empty recycle bin" or some equivalent of that. Only then is the file truly gone forever. However, if you change your mind, and want your file back, you can always dig it out of the recycle bin before you empty it.

You can probably see where this is going:

Danger

When you use `rm` to delete a file or folder, that file or folder is immediately gone **FOREVER!**

Hopefully, you are becoming appropriately terrified of this command. Using it safely requires a little more knowledge than just learning the basic usage. Let's start learning about `rm` directly from its `man` page.

```
NAME
  rm -- remove directory entries

SYNOPSIS
  rm [-dfiPRrvW] file ...

DESCRIPTION
  ...
  -i      Request confirmation before attempting to remove each
         file, regardless of the file's permissions, or whether
         or not the standard input device is a terminal.
         The -i option overrides any previous -f options.

  -f      Attempt to remove the files without prompting for
         confirmation, regardless of the file's permissions.
         If the file does not exist, do not display a diagnostic
         message or modify the exit status to reflect an error.
         The -f option overrides any previous -i options.

  -R      Attempt to remove the file hierarchy rooted in
         each file argument.  If the -i option is specified,
         the user is prompted for confirmation before each directory's
         contents are processed (as well as before the attempt
         is made to remove the directory).  If the user does
         not respond affirmatively, the file hierarchy rooted
         in that directory is skipped.

  -r      Equivalent to -R.

  -v      Be verbose when deleting files, showing them as they are removed.

  ...
```

Simplified, this `man` page is saying that the usage of `rm` goes like this:

```
# delete one or more files
$ rm <file 1> <file 2> ...
```

```
# delete one or more directories and their contents
$ rm -r <dir 1> <dir 2> ...
```

The `-r` flag tells `rm` to "recursively delete" the directory and everything inside of it. `rm` will fail if you try to delete a directory without the `-r` flag. That is for your own safety. It wants to help you not delete more than you want to.

When deleting anything, whether it be files or directories, you can use `-i` to have `bash` double check with you before deleting every file like this:

```
$ rm -i chair.jpeg  
remove chair.jpeg?
```

At that point, only the input `y` or `yes` (along with pressing `Enter ↵`) will result in the file being deleted. If you use `rm -ri some/important/folder`, `rm` would give you this prompt for every file, folder, sub-folder, etc. contained in `some/important/folder`.

You can see how this helps prevent you from catastrophically deleting too much... but at the same time, you can probably sense how annoying this is when the folder you are trying to delete contains thousands of files.

`-f` is short for "force". If you want to cancel out the `-i` flag, you can use the `-f` flag after it, which overrides the `-i` and tries to do a force delete.

So, if you did, `rm -rif some/important/folder`, `rm` would go right ahead and delete the entire folder along with all of its contents without requiring you to confirm any of them.

But wait... why would we ever put `-i` and `-f` into the same call to the `rm` command? Isn't that, like, a waste of typing? Good point. If our goal is to just blow through deleting a folder, we can just type `rm -rf` and be done with it.

However, `rm -rif` is worth talking about. The next two tips show why learning this is not crazy.

Tip #1 - The `alias` Command

A neat `bash` command we have not covered yet is `alias`. `alias` allows you to create command shortcuts for yourself to save you time when you find yourself typing long commands frequently. It works like this:

```
# create a shortcut command called "say-hi"
$ alias say-hi="echo hi"
# use it
$ say-hi
hi
# double check to see how my shortcut works
$ alias say-hi
say-hi='echo hi'
# remove my alias because I don't want it anymore
$ unalias say-hi
```

`alias` can be used in tricky ways. For example, I can make an alias of `echo` that uses the command `echo`. Observe:

```
$ alias echo="echo hi"
$ echo
hi
$ echo bob johnson
hi bob johnson
```

Do you see the trickiness here? Whenever we go to use the `echo` command after creating the `alias echo="echo hi"` alias, `bash` puts all of the arguments we pass to `echo` after the `alias`. So `bash` is really running:

```
echo hi bob johnson
```

The next tip shows how this is relevant to `rm`.


Tip #2 - `rm` is usually an alias! 🙊

Like I said, `rm` is dangerous. `-i` is a way to protect yourself from accidentally deleting too much at once. For this reason, it is super common for `rm` to actually be an alias of `rm -i`! You can check this on your system by running `alias rm`.

This makes it so that, by default, if an inexperienced `bash` user goes to delete an important folder containing one thousand files, they won't make it far before giving up. But *you* know about the `-f` flag, so you can take the safety off and delete *almost* anything.

Notice the emphasis on "almost". **There are many files which your user does not have the necessary permissions to delete by default.** But don't worry, we'll cover the topic of permissions and users in a later chapter so that you can delete those ones, too.

If you took the time to read and understand this section, great job! You are a safe person, and you will handle the destructive power of `rm` responsibly 😎

 **Exercise 3.3**

Check to see if `rm` is an alias on your system. If it isn't, make it an alias for `rm -i`.

[See the solution](#)

⚡ An easy mistake for `bash` noobs to make

Consider how important the Death Star plans were for stopping Darth Vader. Imagine that the rough draft of the Death Star plans were stored on their central Linux mainframe under `rough draft schematics.txt` (naturally the dark side prefers open source tools like Linux).

Once the plans were completed, they made a final version called `schematics.txt`.

So their folder structure looks like this. Notice the unfortunate spaces in one of the file names.

```
schematics
├─ rough draft schematics.txt
└─ schematics.txt
```

Joe-Bob the storm trooper saw that `schematics rough draft.txt` was taking up a lot of space. He figured he could make room more cat videos by getting rid of the rough draft. Luckily, Joe-Bob had learned Linux 3 days before so he knew he didn't need waste his superiors' time by running this idea past them. He ran:

```
$ rm rough draft schematics.txt
```

To Joe-Bob's horror, he saw this output:

```
rm: rough: No such file or directory
rm: draft: No such file or directory
```

Starting to panic now, he checked the contents of `schematics/`.

```
$ ls
rough draft schematics.txt
```

Where is `schematics.txt` ???

Like we have talked about, spaces are the way `bash` tells different arguments apart. So, when Joe-Bob ran `rm rough draft schematics.txt`, `bash` saw three arguments: `rough`, `draft`, and `schematics.txt`. Dutifully, `rm` tried to delete all three of those things as commanded, but it couldn't find `rough` or `draft`. `bash` *did* find `schematics.txt`, though, so it went ahead and deleted it. 1 out of 3 is better than nothing, right?

It would have worked if Joe-Bob had used spaces like this:

```
$ rm "rough draft schematics.txt"
```

But he didn't...

They had to let Joe-Bob go after that incident.

Story Time

Ed Catmull—an amazing computer scientist and co-founder of Pixar—tells this story in his book *Creativity, Inc.* Various versions of the same account can also be found all over the internet.

When *Toy Story 2* was in development, somehow somebody accidentally ran `rm -rf /` on the Linux machine holding *all* of the assets for the movie!

Remember from [Chapter 1](#) that `/` is the root directory? So, `rm -rf /` means: *delete everything* on a machine.

One by one, all of the characters and scenes disappeared right before the production crew's eyes! In a panic, the directors and staff met in a conference room to discuss what to do. They estimated it would take thirty people a year to recreate what had been lost.

After an hour in the meeting, an employee who had been working from home suddenly remembered that she had been creating routine backups of all the production assets. She and one of the technical leaders got into a car and rushed to her home. They grabbed the laptop, wrapped it in blankets, and drove in the slow lane all the way back to Pixar where they were welcomed as heroes by their relieved team.

There are many takeaways you could glean from this story, but mainly: *be careful with `rm`!*

Never run commands you don't understand

Before we discussed `rm`, we had not talked about many parts of Linux or `bash` that you might consider "dangerous".

Sure, with `touch` we could theoretically generate so many empty text files that our computer ran out of space. Or with `mv` you could maybe rename or move a critical file or folder to break something. But those things are reversible and fairly easy to avoid.

However, you should be starting to sense that `bash` can do catastrophic things to your computer that you can not necessarily do from outside the terminal.

So please. For your own safety. **Never run a command in your terminal if you don't understand what it does.** StackOverflow is awesome... but be cautious young one 🙈

⚠ Exercise 3.4

Create the following directory structure. Be sure to keep the spaces in the names.

```
schematics
├─ unnecessary folder/
│   └─ dummy file.txt
├─ rough draft schematics.txt
└─ schematics.txt
```

Now use the `rm` command to delete `dummy file.txt` and THEN `unnecessary folder/`.

After that, remove `rough draft schematics.txt`

[See the solution](#)

Chapter 3 Wrap Up

Fantastic job working your way through this chapter!

You could easily have just googled the `mv` and `rm` commands or read through their `man` pages to see how to use them. But, by reading through this chapter and doing the exercises, you are protected from making catastrophic mistakes using `rm`.

In the next chapter, we will learn about a tool called *wildcards* which take `mv`, `rm`, and several other commands to the next level.

Chapter 3 Exercise Solutions

3.1 - Creating the furniture directory

The directory looks like this:

```
# create the starter files/folders with a flat structure
$ mkdir furniture seats storage
$ touch "bean bag.txt" chair.jpeg couch.jpeg

# verify that they are all there
$ ls
bean bag.txt  chair.jpeg  couch.jpeg  furniture  seats  storage

# move the folders into place
$ mv seats storage furniture

# move the files into place
$ mv "bean bag.txt" chair.jpeg couch.jpeg furniture/seats

# BONUS - clean up the files and folders we made
$ rm -r furniture
```

3.2 - Renaming a file with a bad name

```
# create a file with a bad name
touch "bad filename.txt"

# rename that darn thing
mv "bad filename.txt" "bad-filename.txt"

# BONUS - clean it up
$ rm bad-filename.txt
```

3.3 - Aliasing rm

```
# check to see if rm is already an alias
$ alias rm

# if it is already aliased, unalias it for practice
$ unalias rm

# now give rm a safer alias
$ alias rm="rm -i"

# validate that it worked
$ alias rm
rm='rm -i'

# BONUS - try it out
$ touch dummy-file.txt
$ rm dummy-file.txt
remove dummy-file.txt? y

# dummy-txt should be nowhere to be found
$ ls
```

3.4 - Practicing rm

We want this folder structure:

```
schematics
├─ unnecessary folder/
│   └─ dummy file.txt
├─ rough draft schematics.txt
└─ schematics.txt
```

```
# create the folder structure
$ mkdir schematics
$ mkdir schematics/"unnecessary folder"
$ touch schematics/schematics.txt "schematics/rough draft schematics.txt"
$ touch "schematics/unnecessary folder/dummy file.txt"
```

```
# remove dummy file.txt
$ rm "schematics/unnecessary folder/dummy file.txt"

# remove unnecessary folder
$ rm -r "schematics/unnecessary folder"

# remove rough draft schematics.txt
$ rm "schematics/rough draft schematics.txt"

# validate the result
$ ls schematics
schematics.txt
```

Chapter 4 - Wildcards and Globbing

Introduction

This is going to be a fun chapter 😊. If you have been following along with the exercises, you may have noticed that `bash` commands often take one or more file paths as arguments. Would you agree that it feels tedious to have to type out all of those file paths? I think so.

 **Use the `Tab →` key!**

Okay fine, if you have been using the `Tab →` key like a maniac (as you should!), typing out file path after file path isn't as tedious. But when the number of files gets large, you're going to want a better method.

In this chapter, we will cover an incredibly useful tool called *wildcards* to help us to manipulate huge amounts of files and folders with the greatest of ease 🚀

Let's get started!

Wildcards

In the last chapter we looked at the `rm` and `mv` commands. One highly convenient thing about these two commands is that they can act on several files at once like so:

```
# delete several files
rm <file path 1> <file path 2> <file path 3> ...
```

This is nice, but how much effort does it *really* save us when we use `rm`? We still have to type out every single file path we want to delete.

Or do we?

In `bash`, **wildcards** are special characters that you can include in file paths which allow you to select many files at once. If that definition is confusing, don't worry, we will go over a several examples in this chapter. For now, just know that wildcards are about to become your best friend.

Wildcards in Context

Picture this: you are trying to take a picture on your phone. But every time you hit the camera button, you get a fateful error message saying that your phone is out of storage space 🙄. I know, I know. This is a sensitive topic for all of us.

Well, today is the day to free up that precious phone space by offloading some of those pictures onto your computer. So you do! When you are finished, you have one big soup of pictures in a folder like this:

```
pictures
├── 2021-01-08.HEIC
├── 2021-01-09.HEIC
├── 2021-01-10.HEIC
├── 2021-01-11.jpg
├── 2021-01-12.mp4
├── 2021-01-13.mp4
├── 2021-01-15.mp4
├── 2021-02-08.HEIC
├── 2021-02-09.mp4
├── 2021-02-11.HEIC
├── 2021-02-12.HEIC
├── 2021-02-12.mp4
├── 2021-02-14.HEIC
├── 2021-03-08.HEIC
├── 2021-03-10.HEIC
├── 2021-03-10.jpg
├── 2021-03-12.mov
├── 2021-03-14.HEIC
└── 2021-03-15.jpeg
```

If you are like me, this big, disorganized mass of file names may trigger your OCD. That's okay. My doctor says we can still live perfectly full lives despite our struggles 🤖

Do you know these file types?

In case any of these file extensions are unfamiliar to you:

- `mp4` and `mov` are common video formats
- `jpg` and `jpeg` are equivalent extensions for the JPEG image format
- `HEIC` is an image format often used by Apple devices

Here is the plan. We will clean up this mess by grouping these files into a `videos` folder and a `pictures` folder based on their file type. When we are finished, we will have a folder structure that looks like this:

```
.
├── pictures
│   └── all the pictures...
```

```
└─ videos
  └─ all the videos...
```

We can start by creating the folders. Assume my working directory is `pictures/`.

```
# move to the parent directory of pictures
$ cd ..

# create videos as a sibling directory of pictures
$ mkdir videos

# check the contents of this directory
$ ls
pictures videos
```

Fantastic! Now we just need to move the video files out of `pictures` and into `videos`. Seems simple enough. BUT! With the method we have been using to move files so far, i.e. typing out the name of every file by hand, this would be sooo tedious.

I'll write it out for you just this once, but please don't ever ask me to do this again.

```
# move into pictures so we won't have to prefix all
# of the video file paths with "pictures/"
$ cd pictures

# get busy movin' or get busy dyin'
$ mv 2021-01-12.mp4 2021-01-13.mp4 2021-01-15.mp4 2021-02-09.mp4 2021-02-12.mp4 2021-03-12.mov ../videos
```

Ugh, that was awful. Imagine how much typing that would have taken if there had been thousands of files 🤖. Luckily, there is a better way.

Behold!

```
# a more excellent way
$ cd pictures
$ mv *.mp4 *.mov ../videos
```

DONE!

That is so much shorter! Not to mention, I didn't even have to know any of the specific names of the files. Let's talk about this.

Wildcard 1 - * (asterisk)

`*` is a wildcard. That means that when we use it in a command, it "expands" and turns into any character that it "matches". The best way to understand `*` is by looking at some examples. We can use the `echo` command to see what various wildcards expand to.

Consider this directory called `example`.

```
example
├─ a.txt
├─ b.txt
├─ c.txt
├─ a.jpg
└─ b.jpeg
```

Assume `example` is our working directory. Let's see what this wildcard expands to used in a few different ways.

```
# match any file or folder name
$ echo *
a.txt b.txt c.txt a.jpg b.jpeg

# match any file or folder names ending with ".txt"
$ echo *.txt
a.txt b.txt c.txt

# match any file or folder names starting with "a"
$ echo a*
a.txt a.jpg

# match any file or folder names containing "jp"
$ echo *jp*
a.jpg b.jpeg
```

Now that you have seen some examples, let's define `*` one more time. `*` means: match any number of any character. An important term to know here is *pattern*. A `pattern` is a combination of wildcards and regular characters.

`*`, `.jpg`, `*.jpg`, and `*jp*` are all examples of patterns. Even normal file paths are patterns; they just don't have any wildcards. When you use a pattern in a `bash` command, `bash` tries to match as many file paths with that pattern as it can, and then it substitutes those in place of the pattern.

So, inside of the `example` directory, when we run `echo *`, the `*` pattern expands to every file it can find there. So, `bash` ends up running `echo a.txt b.txt c.txt a.jpg b.jpeg`. Naturally, because of the spaces, each of those file paths gets passed to `echo` as a separate argument.

⚠ Exercise 4.1

Use this command to create files with the same names as the images and videos we saw in the first example of the chapter.

```
$ touch 2021-01-08.HEIC 2021-01-09.HEIC 2021-01-10.HEIC 2021-01-11.jpg
2021-01-12.mp4 2021-01-13.mp4 2021-01-15.mp4 2021-02-08.HEIC 2021-02-
09.mp4 2021-02-11.HEIC 2021-02-12.HEIC 2021-02-12.mp4 2021-02-14.HEIC
2021-03-08.HEIC 2021-03-10.HEIC 2021-03-10.jpg 2021-03-12.mov 2021-03-
14.HEIC 2021-03-15.jpeg
```

Use `mkdir`, `mv`, and the `*` wildcard to organize these files by file type like this:

```
.
├── pictures
│   ├── heic
│   │   └── all the HEIC files...
│   └── jpeg
│       └── all the jpeg and jpg files...
└── videos
    ├── mp4
    │   └── all the mp4 files...
    └── mov
        └── all the mov files...
```

You can use `ls -R directory/containing/pictures/and/videos` to check your answer.

[See the solution](#)

⚡ Wildcards hate spaces, too! 😞

Suppose you have a file named `silly file with spaces.txt`. You want to delete this file using `rm` – along with many other `.txt` files.

```
rm *.txt will become rm silly file with spaces.txt.
```

Well, `rm` will see all of these as separate arguments: `silly`, `file`, `with`, and `spaces.txt`. In the best case, no files with those names exist in your working directory. In the worst case, one or all of `silly`, `file`, and `with` *did* exist... and you just deleted them permanently.

Advanced Tip about **/*

We have seen `*` matches any number of any character **in the particular folder where you invoke** `*`. But if you want to see every subfolder and its contents using wildcards, you can use `**/*`.

For example, after doing **Exercise 4.1**, we can get a list of every folder and subfolder in `pictures` and `videos` by running `echo **/*` from the parent directory of those two folders.

If you tried this on the root directory like this: `echo /**/*`, you would get an error saying that you are trying to pass too many arguments to `echo` ... because there are a *ton* of files on your computer.

To be honest, while there are technically three wildcards in `bash` most people only use `*` in their daily work with the terminal.

However, for the sake of being thorough, we will cover the other two now.

Wildcard 2 - ?

In contrast with `*`, which can match *any number* of any character, `?` is a wildcard that can match any character, but only *one*. Again, we will need to see some examples for this to make sense.

Let's use a new `example` directory for this.

```
example
├─ aaa.jpg
├─ bbb.jpg
├─ ccc.jpeg
├─ aaa.txt
├─ bbb.txt
└─ a.txt
```

Here are a few patterns using `?`

```
# match file names that start with 3 characters and end with ".jpg"
$ echo ??? .jpg
aaa.jpg bbb.jpg

# match file names that start with "aaa." and end with exactly 3 characters
$ echo aaa.???
aaa.jpg aaa.txt

# match any file name that is 7 characters long
$ echo ??????
aaa.jpg bbb.jpg aaa.txt bbb.txt
```


Hopefully you get the picture: `?` is used when you want to control the length of the file names matched by a pattern.

Of course, you can combine `*` and `?` in one pattern, in which case the length of the matched file names would not be fixed. For example, `*.???` would match all files whose extension is three characters long... so no `.py` files 😞

Alright, let's move onto the last wildcard!

Wildcard 3 - `[]`

The last wildcard is `[]`. This wildcard is similar to `?` in that it only matches one character at a time. The difference is that with `[]` we can choose which characters we want to match.

For example,

Pattern	Description
<code>[a]</code>	Only the letter "a"
<code>[abc]</code>	Any of the letters "a", "b", or "c"
<code>[a-c]</code>	Any of the letters "a", "b", or "c"
<code>[a-zA-z0-9]</code>	Any capital letter, lowercase letter, or number

As you can see from this table, there are two main ways to define `[]` character sets in patterns.

1. We can manually type out all of the characters we want to match, e.g. `[abcde]`
2. We can use a hyphen (`-`) as a shorthand for a range of characters to match, e.g. `[a-e]`

Let's look at some examples of pattern matching using the `[]` wildcard. This `example` folder will be our working directory.

```
example
├── aaa.jpg
├── bbb.jpg
├── ccc.jpeg
├── aaa.txt
├── bbb.txt
└── a.txt
```

And here are the pattern match examples:

```
# match ".jpg" file names where the first three letters are one of "a", "b",
or "c"
$ echo [abc][abc][abc].jpg
aaa.jpg bbb.jpg

# match all file names beginning with "a" or "c"
$ echo [ac]*
a.txt aaa.jpg aaa.txt ccc.jpeg

# match all file names containing "a" or "b" anywhere in them
$ echo *[ab]*
a.txt aaa.jpg aaa.txt bbb.jpg bbb.txt
```

If you read through this section and the two before it, you have learned all three wildcards! We will close this chapter with some final points about their general usage.

Wildcards and Double Quotes

You can use wildcards anywhere in a file path. However, if the wildcard is inside of quotation marks, `bash` will not expand it. For example:

```
# will not expand
$ echo "/Users/eric/*"
/Users/eric/*

# will expand
$ echo "/Users/eric/"*
... everything directly inside of /Users/eric ...
```

Globbering in Python!

Pattern matching using wildcards is called `globbing`. This term comes from the idea that when we use wildcard-powered search patterns, we are doing a "*global* search" for files matching that pattern.

There is a built-in `glob` library in Python for doing the same thing!

Assume this example folder is placed at my user directory at `/Users/eric/example`

```
example
├── a.jpg
└── b.jpeg
```

In two lines of code, we can leverage the power and flexibility of wildcards to get the absolute file paths of every file in `example/`.

```
>>> from glob import glob
>>> jpeg_files = glob("/Users/eric/example/*")
>>> print(jpeg_files)
["/Users/eric/example/a.jpg", "/Users/eric/example/b.jpeg"]
>>> ... proceed to do some awesome image processing ...
```

Not bad, right?

In Python, code can get complicated very quickly when you are trying to search for specific files in the file tree.

However, often a simple "glob pattern" is all you need when searching for files in code. In those cases, if you can use `glob`, you get massive style points for having simple, readable code 😊.

Chapter 4 Wrap Up

There you have it! You have learned the three `bash` wildcards and how to use them in `bash` and Python. Let's summarize them now:

- `*` - Matches any number of any character
- `?` - Matches one character (but it can be any character)
- `[]` - Matches one character from a character set we define

All of these are quite useful. You can use them for commands like `mv` and `rm` to manipulate several files at once. Do not stress too much about memorizing `?` and `[]` as they do not tend to come up as often as `*`. Please *do* memorize the `*` wildcard.

Notice that, in this chapter, we did not cover any new commands. Rather, we discussed a tool that helps us use *many* `bash` commands more effectively. In the next chapter, we *will* see a few new commands, but the main focus will be yet another *awesome* tool that enhances almost all `bash` commands. See you there!

Chapter 4 Exercise Solutions

4.1 - Sorting Files by Type with Wildcards

The goal is to achieve this folder structure:

```
.
├── pictures
│   ├── heic
│   │   └── all the HEIC files...
│   └── jpeg
│       └── all the jpeg and jpg files...
└── videos
```

```
├─ mp4
│   └─ all the mp4 files...
├─ mov
│   └─ all the mov files...
```

```
# create the mass of files
$ touch 2021-01-08.HEIC 2021-01-09.HEIC 2021-01-10.HEIC 2021-01-11.jpg 2021-
01-12.mp4 2021-01-13.mp4 2021-01-15.mp4 2021-02-08.HEIC 2021-02-09.mp4 2021-
02-11.HEIC 2021-02-12.HEIC 2021-02-12.mp4 2021-02-14.HEIC 2021-03-08.HEIC
2021-03-10.HEIC 2021-03-10.jpg 2021-03-12.mov 2021-03-14.HEIC 2021-03-15.jpeg

# create the directories
$ mkdir pictures videos
$ mkdir pictures/heic pictures/jpeg
$ mkdir videos/mp4 videos/mov

# move the files into their folders
$ mv *jp* pictures/jpeg # matches ".jpg" and ".jpeg" files
$ mv *.HEIC pictures/heic
$ mv *.mov videos/mov
$ mv *.mp4 videos/mp4
```

The end result should be this:

```
.
├─ pictures
│   ├── heic
│   │   ├── 2021-01-08.HEIC
│   │   ├── 2021-01-09.HEIC
│   │   ├── 2021-01-10.HEIC
│   │   ├── 2021-02-08.HEIC
│   │   ├── 2021-02-11.HEIC
│   │   ├── 2021-02-12.HEIC
│   │   ├── 2021-02-14.HEIC
│   │   ├── 2021-03-08.HEIC
│   │   ├── 2021-03-10.HEIC
│   │   └─ 2021-03-14.HEIC
│   └─ jpeg
│       ├── 2021-01-11.jpg
│       ├── 2021-03-10.jpg
│       └─ 2021-03-15.jpeg
└─ videos
    ├── mov
    │   └─ 2021-03-12.mov
    └─ mp4
        ├── 2021-01-12.mp4
        ├── 2021-01-13.mp4
        ├── 2021-01-15.mp4
        ├── 2021-02-09.mp4
        └─ 2021-02-12.mp4
```

Chapter 5 - The Power of Redirection

Introduction

In the last chapter, we talked about wildcards, and we saw how they can save us a lot of manual effort when trying to manipulate several files at once. A great thing about wildcards is that you can use them with tons of different commands.

In this chapter, we will explore another awesome tool called "redirection operators" which, similar to wildcards, will supercharge what you can do with almost every `bash` command.

Essentially, redirection operators allow you to chain `bash` commands together by passing (or *redirecting*) the output of one command into another as input. This is powerful stuff!

We will also pick up some super useful `bash` commands along the way.

Let's get started!

A First Encounter with Redirection Operators

For certain commands, when you run them in the terminal, we get a bunch of text as output. For example,

```
$ echo "hi"  
hi
```

You: Wait, didn't we talk about `echo`, like, fifty chapters ago?

Me: Yes! But now we need to get philosophical. *Why* do we see `hi` printed back out at us in the terminal?

I can tell you're unimpressed with this question. In order to explain why this question is important, I must introduce you to a fantastic new command: `cat`.

INSERT PUBLIC DOMAIN PICTURE OF CAT HERE

`cat` outputs the contents of a file to the terminal. The basic usage of `cat` goes like this:

```
# print the contents of a file
$ cat <file path>
```

Note

Technically `cat` is short for "con `cat` enate" because it can be used to concatenate the contents of multiple files. So, really the usage is like this:

```
# print the contents of multiple files one after another
$ cat <file path 1> <file path 2> ...
```

Alright, alright. So, now you know `cat`. Let's get back to `echo`. Observe:

```
# redirect "hi"; note that "hi" isn't getting printed out!
$ echo "hi" > greeting.txt

# something must be wrong! Let's look around
$ ls
greeting.txt

# what's this?? Let's look inside of this file
$ cat greeting.txt
hi
```

Can you see what just happened?

We ran `echo "hi"`, but we did not see the usual `hi` output in the terminal. *Instead*, the output of `echo "hi"` AKA `hi` was *redirected* into a text file called `greeting.txt` by a redirection operator, `>`.

To understand what "redirecting output" means, we need to closer look at what the inputs and outputs of `bash` commands really are. We will break that down in the next section.

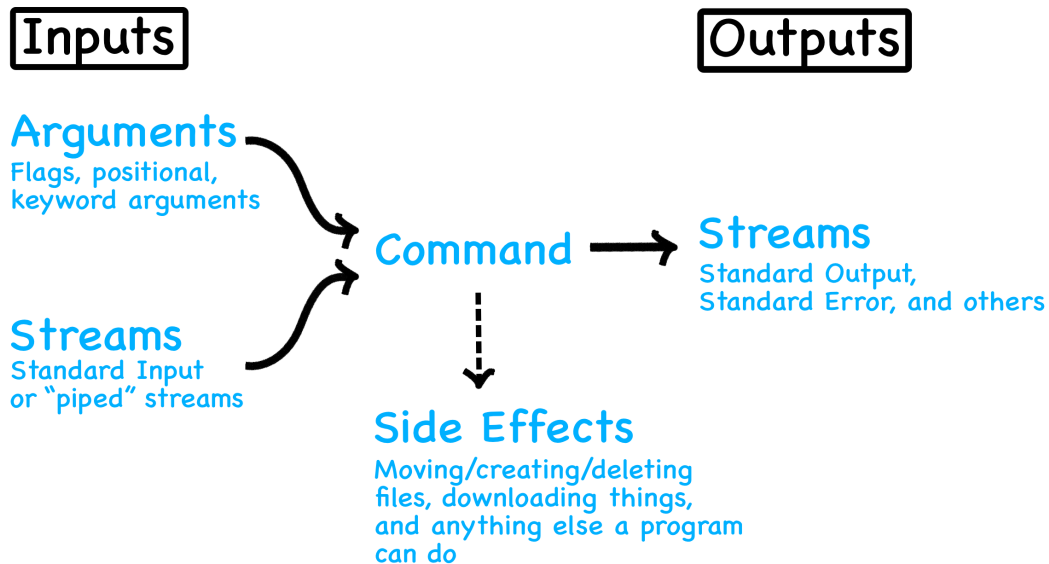
Streams - The Communication Channels of `bash` Commands

In the last section, you saw your first redirection operator: `>`. In our example, we redirected the output of `echo` into a text file. But what do we mean by "output"?

All `bash` commands have inputs and/or outputs. We have already seen one example of an input: command arguments. But there is another type of input: streams. The outputs of `bash` commands can actually be streams as well.

Here's a diagram to help visualize this. Don't worry if you don't recognize the names "standard in" and "standard out". We will be covering those soon!

How to Think About Commands



So what are streams?

In Linux, **streams** are data (such as text) that travel from one process or command to another.

To make this definition more tangible, imagine a stream as a file somewhere on your machine. When `echo` executes, it "writes" its output to that imaginary file (the stream). Then, other commands (or we) can take that data and do something with it if we want.

It turns out, this is almost exactly what is happening. There is a stream called "standard output" or **stdout** for short. By default, all commands that have text as output write their output to `stdout`. If we don't explicitly do something with the data written to `stdout`, then, by default, `bash` just prints it to the terminal.

This is where redirection operators come in! **Redirection operators** or **I/O redirection operators** (I/O is short for input/output) are special symbols we can use to tell `bash` what to do with data in streams.

In the case of our first example, we used `>` to redirect the string `"hi"`, which was sitting in the `stdout` stream into a text file, `greeting.txt`.

Here are all of the I/O redirection operators. There will be examples of all of these later, so don't worry if these explanations don't make sense right away.

Operator	Description
----------	-------------

Operator	Description
>	Redirect a stream to text file. If the text file does not already exist, create it. If it does exist, *overwrite* it.
>>	Append the contents of a stream to a text file if it already exists. Otherwise, create it.
(pipe)	Redirect or "pipe" a stream into a command to be used as an input. Note, the " " symbol is on the "\" (backslash) key on the keyboard.
<	Send the contents of an existing text file to a stream to be used as input for a command.
<<	Write out several lines of text and send it to a command as a stream without actually creating a file with those lines on disk.

Enough words! Let's do some examples!

Creating & Overwriting Files with >

```
# write some text to example.txt; create it if it does not already exist
$ echo "some text" > example.txt
$ echo "some more text" > example.txt
$ echo "some final text" > example.txt

# send the contents of example.txt to stdout
$ cat example.txt
some final text
```

Only `some final text` ended up in `example.txt` because `>` overwrites the target file if it already exists.

Creating & Appending to Files with >>

```
# write some text to example.txt; create it if it does not already exist
$ echo "some text" >> example.txt
$ echo "some more text" >> example.txt
$ echo "some final text" >> example.txt

# send the contents of example.txt to stdout
$ cat example.txt
some text
some more text
some final text
```


All three lines ended up in `example.txt` because `>>` appends to the end of a file if it already exists.

Exercise 5.1

Use `>` and `>>` to capture the output of `ls` in a file, but first, use `echo` to give that file a title.

Hint, `/bin` has a good amount of things in it.

[See the solution](#)

In these examples, we took the **output** of various commands from the `stdout` stream and redirected it into a file. In the next section, we will look at using streams as **inputs** for commands as well.

Taking Streams as **bash** Command Inputs

We have seen that **bash** commands often write output to a stream, `stdout`. Many **bash** commands can take a stream as input as well.

It turns out, `cat` can!

We saw earlier that the typical usage of `cat` is `cat <file 1> <file 2> ...`. However, if you try using `cat` with *no arguments*, then `cat` expects a stream as input instead of arguments.

Note

For **bash** commands, taking arguments as input and taking streams as inputs are not mutually exclusive. Many commands can take both at the same time.

Check it out!

```
# create a text file with some text
echo "radical, duude" > cool.txt

# send the contents of the text file to cat as a stream
cat < cool.txt
radical, duude
```

We just redirected the contents of `cool.txt` to a stream and then sent them into `cat` as input. Nice!

Standard Input

Now, watch what happens when we run `cat` with no arguments so that it expects a stream, *but we don't give it one*.

```
# the cat command hangs
$ cat
```

The command just **hangs** (seems to run forever without completing). What does this mean? Is our terminal going to be taken up by this hanging `cat` command forever??

Actually, `cat` is waiting for us!

As we've seen, there is a default output stream called "standard output" that gets written to the terminal by default.

Very similarly, there is also a default *input* stream called "standard input" that gets used by default if we don't feed a command a stream as input when it expects one. We call this default input stream **stdin**.

It turns out, *stdin* is connected to our keyboard!

So if we just run `cat` with no arguments, and start typing things into the terminal, `cat` will repeat back to us everything we type.

```
$ cat
stop mimicking me!
stop mimicking me!
no!
no!
I am dumb!
You are dumb! <-- not actually real
^C
```

Tip

Use `^ Ctrl + C` to kill a command when it is hanging like this (or running too long)

More Examples of Using Streams as an Input

Now that we know that `cat` can take a stream as input, let's do some quick examples of each of the input-stream operators: `<<`, `<`, and `|`, and then we'll go over some of the most useful commands involving streams as a reward for learning all of this stream theory.

The `<` Operator

We already saw this earlier in the `cool.txt` example. We can send the contents of a file to a stream to be used as input for a command using `<`.

The << Operator

The << operator lets us write several lines of text to a stream to be used as input to a command *without* actually writing those lines as a file on disk. So, whereas <, relies on a file to already be present, << lets us skip the step of creating a file.

It does this by letting us write a **heredoc**, or a "document right here in the command line" 😊

In action, it looks like this:

```
$ cat << EOF
heredoc> I
heredoc> AM
heredoc> A
heredoc> WIZARD!!!
heredoc> EOF
I
AM
A
WIZARD!
```

Note

Whenever you see a terminal line starting with **something>**, it means the terminal is waiting for you to finish entering a string that you started to type.

For example, if you type a double-quote `"`, you will get a **dquote>** prompt. This allows you to type a long, multi-line string right in the terminal. **bash** quits out of that if you type a second `"`, which it considers to be the closing double quote.

The same thing happens if you type `\`, `'`, `(`, `{`, ``` (backtick), or any other "opening" symbol that should have an accompanying "closing" symbol.

As usual, if you don't feel like completing your string, you can use `^C` to kill the **bash** process instead.

EOF is an arbitrary string I chose. **EOF** stands for "End Of File" and it is pretty common to use when you write heredocs. To write a heredoc, you just need to choose *some* string to let the heredoc tool know when you have finished typing your multi-line string.

So I could just as easily have done this,

```
$ cat << GUMMY_SNACKS
heredoc> I
heredoc> AM
heredoc> A
heredoc> WIZARD!!!
heredoc> GUMMY_SNACKS
I
AM
```

```
A
WIZARD!
```

but I was trying to look legit', you know? 😊

I admit that just spewing all of this output into the terminal is not very useful. Let's redirect it to a file instead. The placement of the `>` redirection operator is a little weird here:

```
# write this multi-line string to a file
$ cat << GUMMY_SNACKS > solemn-declaration.txt
heredoc> I
heredoc> AM
heredoc> A
heredoc> WIZARD!!!
heredoc> GUMMY_SNACKS

# inspect the file
$ cat solemn-declaration.txt
I
AM
A
WIZARD!!!
```

Note

We could also have created this multi-line file like this:

```
# use > for the first line in case this file already exists
$ echo "I" > solemn-declaration.txt
# append the subsequent lines
$ echo "AM" >> solemn-declaration.txt
$ echo "A" >> solemn-declaration.txt
$ echo "WIZARD!!!" >> solemn-declaration.txt
```

This doesn't look quite as clean, but it works, and you should not feel bad if you use it!

The | (Pipe) Operator

The `|` operator is pretty straightforward. It takes the output of one command and forwards it (or `pipes` it) into another command as an input stream. For example,

```
# overcomplicated way of printing "hi"
$ echo "hi" | cat
hi
```

And that is all of the I/O redirection operators! You deserve better examples than `echo "hi" | cat`, but we haven't seen any commands besides `cat` that take streams as inputs. In the next section, you will be rewarded for your diligence and learn some sweet new commands!

Awesome Commands that Take Streams as Input

As promised, here are several fun commands that you can pipe streams into.

grep - The `^ Ctrl` + `F` of bash

`grep` is short for "global regular expression print". Totally fascinating, I know. But actually, it is!

`grep` is a tool for doing *global* searches on files and streams using a pattern matching language called "regular expressions" (or `regex` for short).

In the last chapter, we used wildcards to create patterns to search for files whose names matched those patterns.

Similarly, Regex is an *extremely* powerful language for writing search patterns to find specific substrings in large amounts of text. In short, it is an overpowered `^ Ctrl` + `F` for all programming/scripting languages including `bash` 😊

Going into the details of Regex is outside the scope of this book, but I *highly* recommend that you find some tutorials online and learn it, as it is a rite of passage for all coders.

<https://regexr.com/> is a great place to practice regular expressions and test ones you have already written.

Suffice it to say that `friend` is a valid regular expression to find any instances of `friend` in a document, but regex can do a whole lot more.

The basic usage of `grep` is this,

```
# show all lines matching a regular expression in one or more files
$ grep <regular expression> <file 1> <file 2> ...

# show all lines matching a regular expression in a stream
$ grep <regular expression>
```

The stream version is the one we will use. Ahem,

```
# create a text file called sick.txt
$ cat << EOF > sick.txt
heredoc> If I were a rich
heredoc> man... doo be doo be...
heredoc> would it spoil some
heredoc> vast eternal plan if I
heredoc> were a wealthy man?
heredoc> EOF

# search for all lines containing "I"
$ grep "I" sick.txt
If I were a rich
vast eternal plan if I
```

```
# search for all lines containing "I" (using a stream)
$ cat sick.txt | grep "I"
If I were a rich
vast eternal plan if I
```

There you have it! `^ Ctrl` + `F` is now in the terminal 😊

Tip

You could also use this with `ls` if a folder has TONS of contents, and you want to find a specific file.

Or you could use it with `pip freeze` if you have many Python packages installed, and want to check for a specific one.

Exercise 5.2

Use `grep` on the output of `ls` to see if your `/bin` folder has any files containing the letter `d`.

[See the solution](#)

pbcopy and pbpaste

Note for non-MacOS Users

Sadly, these amazing commands are not available natively for the Windows Linux Subsystem or Ubuntu. They *can* be set up, but it takes some effort.

Try the [instructions in the appendix](#) if you would like to try to set them up for yourself. You should know:

- These commands are non-standard in the WSL and Ubuntu, so at some point these instructions may go out of date.
- The setup uses material that we will discuss in the later chapters on writing `bash` scripts, so you may want to wait until then and come back here.

<https://www.techtronic.us/pbcopy-pbpaste-for-wsl/>

pbcopy

You can copy streams directly to your clipboard and paste them anywhere both inside and outside of the terminal!

```
# put some text into the clipboard
cat complicated-url-to-awesome-site.txt | pbcopy
```

You could do something like this and paste it straight into your browser 🚀

pbpaste

If some text from the web or `pbcopy` is already in your clipboard, you can access it as a stream with `pbpaste` like this,

```
# paste the contents of your clipboard to a new file
$ pbpaste > something-awesome.txt

# append the contents of your clipboard to an existing file
$ pbpaste >> something-awesome.txt

# pass the contents of your clipboard as an input stream
$ pbpaste | cat
something really awesome
```

Try it out!

n1

`n1` is short for "number lines". It takes a filename or a stream and adds line numbers to the beginning of the line.

This can be useful when you are troubleshooting error messages on a forum or in a video call. You can pipe the error message into the `n1` command to make it easier for people to reference specific lines in the error to discuss.

For example,

```
$ cat << ERROR_MSG | n1
heredoc> NameError      Traceback (most recent call last)
heredoc> <ipython-input-1-141b3ea3f03f> in <module>
heredoc>
heredoc> NameError: name 'k' is not defined
heredoc> ERROR_MSG

1 NameError      Traceback (most recent call last)
2 <ipython-input-1-141b3ea3f03f> in <module>
3 ----> 1 k

4 NameError: name 'k' is not defined
```

If you have `pbcopy` / `pbpaste` working, you can simply copy the error message in your terminal and get the same result with,

```
$ pbpaste | n1
```

sort

`sort` can sort the lines of a stream or file in a variety of ways. By default, it alphabetizes them.

It accepts many flags and keyword arguments to do things like sort in reverse order (`--reverse`), check if the lines in the stream are already sorted (`--check`), etc.

```
# sort the lines in this stream and add line numbers
$ cat << TEXT | sort | nl
heredoc> Hello,
heredoc> my
heredoc> friends!
heredoc> TEXT

1 Hello,
2 friends!
3 my
```

wc

`wc` is short for "word count". By default, it counts the number of lines, words, and characters in a file or stream.

```
$ cat << TEXT > some-text.txt
heredoc> Hello,
heredoc> my dear
heredoc> friends!
heredoc> TEXT
$ wc < some-text.txt
3      4     24
```

This is saying that the stream contains:

- 3 lines,
- 4 words, and
- 24 characters

Exercise 5.3

Take some time to play with `wc`, `sort`, and `nl`.

Chapter 5 Wrap Up

Great work for making it to the end of another dense chapter. In this chapter we saw that `bash` commands often use streams as inputs and outputs.

With that knowledge, you can do a lot more with the command output than stare at it in the terminal. You can store it in a file or forward it into other commands.

I want to apologize if the `wc`, `sort`, and `n1` examples were boring. (But `grep`, `pbcopy`, and `pbpaste` are legitimately awesome)

It may not immediately be obvious how these particular commands will be useful to you down the road. In reality, you will most often use redirection operators to read and write to files.

The main reason we couldn't go over more exciting examples is that most of the coolest `bash` commands do not come with Linux out of the box. You have to install them! We will learn how to do that soon, I promise, and then this knowledge will pay off.

In the mean time, continue on to the next chapter to learn *one final trick* for combining commands using variables.

After you learn that, you will finally be ready to learn how to download new commands and even write your own `bash` scripts! And then we will be able to move on to how this all ties into making you an awesome Python developer.

So with that motivation, read on!

Chapter 5 Exercise Solutions

5.1 - Capturing the Output of `ls` with Redirection Operators

```
# create a file and give it a title
$ echo "The Contents of /bin" > bin.txt
# append the contents of /bin to the file; use >> instead of > so we don't
  overwrite bin.txt
$ ls /bin >> bin.txt
# check that it worked correctly
$ cat bin.txt
...
```

5.2 - "grepping" `/bin`

```
# search /bin for any files or folders containing the letter "d"
$ ls /bin | grep d
```

5.3 - Playing with `n1`, `wc`, and `sort`

There is no official solution for this one. You can pipe the output of `ls` or `cat` into each of these.