Better Git Started

An Introduction to the Command Line, GitHub, and Git

Better Git Started
An Introduction to the Command Line, GitHub, and Git

Ian Curtis

Published: September 2, 2022

Last Updated: May 4, 2023

To my family, who didn't even know I was writing a book and who was very surprised when I said I finished it.

Acknowledgements

---

[2]`miro.com/`

Publisher's Note

About this Edition. *Better Git Started: An Introduction to the Command Line, GitHub, and Git* was written by Ian Curtis, a senior at Grand Valley State University and Editorial Assistant for the GVSU Libraries. This text was developed as part of the Accelerating Open Educational Resources Initiative at Grand Valley State University[3], with support from the University Libraries and the President's Innovation Fund.

This edition is released under a Creative Commons - Attribution - Sharealike license (CC-BY-SA 4.0[4]). This allows users to use, share, and adapt the work as long as they provide attribution to the creator(s) and apply the same license to any derivative work or adaptation.

The code used to generate this book can be found in the author's GitHub repository[5]. This includes the PreTeXt XML files, figures, customization files (XSL), and the output (HTML and PDF). Contributors are encouraged to submit an issue or a pull request with changes, especially regarding accessibility (as mentioned below).

Suggested Attribution: *Better Git Started: An Introduction to the Command Line, GitHub, and Git*, by Ian Curtis, published by the Grand Valley State University Libraries, 2022.

Accessibility Statement. The Grand Valley State University Libraries strive to ensure our tools, devices, services, and environments are available to and usable by as many people as possible.

The web version of *Better Git Started* incorporates the following features to support accessibility:

- All content can be navigated by use of a keyboard

- Links, headings, and tables have been designed to work with screen readers

- Code in PreTeXt is rendered with separate blocks so they can be understood by using screen readers and/or other assistive devices.

---

[3]gvsu.edu/library/sc/AcceleratingOER
[4]https://creativecommons.org/licenses/by-sa/4.0/
[5]github.com/ian-curtis/gitstarted

How To Use This Book

Congratulations on deciding to explore the command line! The ideas explained in this book are quite useful, espeically if you plan to go into a coding career, need to store code files publicly, and/or need some version control. This book is designed for those who have no experience with Git, GitHub, or the command line. I assume that you have never touched your terminal (or even know what one is). I don't assume you have a GitHub account or know anything about how to use Git.

Git is an incredibly powerful tool, but it does come with a learning curve. While there are text editors and interfaces out there that make using Git somewhat easier, I believe that it is best to learn the Git essentials the "harder" way. I hope that in doing so you will be able to better understand the steps in the Git process and that using the features in text editors will be easier. Rather than just memorizing how to use a text editor or the commands to enter, I take a more direct approach and show you Git on the command line to help you understand the process. Once you have that down, you can apply the same skills to a variety of different interfaces as you explore new software and resources.

The book is broken up into a few pieces:

1. Multiple prefaces explaining necessary materials depending on your operating system (Materials: Windows, p. xiii, Materials: Mac, p. xv, Materials: Linux, p. xvii)

2. Introductory materials introducing general computer basics, etiquette, and recommended practices (Chapter 1, p. 3) and an introduction to the command line (Chapter 2, p. 15)

3. A detailed introduction and get-started guide to GitHub (Chapter 3, p. 27) and Git (Chapter 4, p. 45) when working alone

4. A detailed introduction and get-started guide to GitHub (Chapter 5, p. 63) and Git (Chapter 6, p. 71) when working with collaborators

5. Appendices with some bonus features, resources, and troubleshooting

The best way to use this book is to first determine where you are in terms of knowledge. Do you know anything about the terminal? If so, you might consider jumping to Chapter 3, p. 27. Do you know about GitHub, but not the terminal? Perhaps start with Chapter 2, p. 15 then move to Chapter 4, p. 45. Regardless of the path you choose, I highly recommend at least scanning the sections in Chapter 1, p. 3. I do assume you have an idea of the topics in that chapter and hope you at least consider adopting some of the recommended practices. If you have no idea what anything is in this book, start at the beginning and just read consecutively.

That being said, be aware that skipping chapters or sections may cause a disconnect between the book activities and your participation. Most sections have activities scattered

throughout and some chapters even have a lengthy extensive activity at their ends and these activites build off of one another. Even if you want to skip chapters, it might be worth it to take the time to participate in these activities along the way. If you are unfamiliar with the content, be sure to follow along with the activities.

The book is designed to be interactive where you follow along as you read; however, it is not a traditional textbook filled with exercises, definitions, and theorems. I am not testing you on any of this as I believe that Git and GitHub are skills best learned by doing, not through an exam. Feel free to make mistakes, break things, and revist chapters you forgot or didn't understand completely. Answers to activities and exercises can be found in Appendix D, p. 97.

As you read, if anything starts to become too confusing or overwhelming, take a break. While a dedicated reader might be able to read this book and complete the activities in one day, it is not recommended to do so. There is quite a bit of new content here and too much at once will certainly lead to an information overload. So take your time, read carefully, and, again, don't be afraid to reread parts of the book anytime you need a refresher on things. If you think that something deserves more attention and more explanation, feel free to open an issue on GitHub (don't know how? see Section B.1, p. 87) and let me know.

If you are prepared to dive in, visit the following prefaces to download the necessary materials, then proceed to learning!

Known Issues. There are certain parts of the book that could be better. I list some below.

- Some figures, particularly the branch diagrams, are blurry from file conversion.

- Images could probably use better alt text

- Certain parts of the appendices need more content for Windows and Linux users

- As I am not a professional index creator, some entries may need to be moved around, some may need to be added, and/or other edits

Materials: Windows

As this book was written on the assumption that you have no experience with Git or the command line, it also assumes you do not have necessary materials (software). Downloads will differ slightly depending on it you have a Mac or a PC and if you are running Linux. The following will provide necessary materials to use this book as well as links and some relevant instructions *for those who are using Windows.* If you are running Windows Subsystem for Linux, instructions might be slightly different; however, start here and jump to a quick Google search if something goes wrong.

A Text Editor. In order to edit files as demonstrated in this book, you will need a text editor outside of the basic default editor. Any editor will work such as Sublime Text[6].

If you are already familiar with an editor or if you like a particular one, use it! However, I personally prefer Visual Studio Code[7] which can be downloaded at the previous link. Throughout this book, I will refer to VS Code with some guidelines on how to find certain features. Should you decide to use another editor, it is up to you to learn how to use it. If you are a beginner with Git and the command line, I would stick with VS Code although as you gain experience you might think about switching to Sublime Text.

Git/Git Bash. To use Git, you need to download and install it on your computer. Depending on your computer and where you got it, you may actually already have Git (for instance, university laptops). To run a quick check, use `Windows` + `S` and search for "git bash". If an application pops up, great! You do not need to install anything. If not, carry on.

Windows does come with its own command prompt. (What's a command prompt? See Section 2.1, p. 15.) However, its syntax doesn't match other terminals and notably is different from Macs. Thus, to ensure that everyone is on the same page, we will download a piece of software called Git Bash that allows everyone to use the same commands.

To install Git Bash, follow these next steps. Yuvraj Chandra wrote a useful Git Bash installation guide[8] that may help you if you get stuck.

1. Visit the Git website[9] and select the download for Windows.

2. Open the `.exe` file and follow the instructions to install Git Bash. Most of the default options should not be changed. However, please read the following notes before installing.

---

[6]`sublimetext.com/download`
[7]`code.visualstudio.com/Download`
[8]`makeuseof.com/install-git-git-bash-windows/`
[9]`git-scm.com/downloads`

- Your computer may ask you if you want to allow the app to modify your computer; it is safe to say yes.

- When you reach the "Select Components" screen, I recommend leaving the default boxes checked as they will be useful later on.

- When you reach the "Choosing the default editor used by Git" screen, I recommend selecting the "Use Visual Studio Code..." option (if you have decided to use VS Code).

- When you reach the "Adjusting the name of the initial branch..." screen, I recommend selecting "Override the default branch name..." and type in `main` (lowercase) into the box. This will help us match the Mac version and respect Git's attempts to be more inclusive[10].

GitHub. You will also need a GitHub account. Since this is an extensive topic, this is covered in Subsection 3.1.1, p. 27.

---

[10] theserverside.com/feature/Why-GitHub-renamed-its-master-branch-to-main

Materials: Mac

As this book was written on the assumption that you have no experience with Git or the command line, it also assumes you do not have necessary materials (software). Downloads will differ slightly depending on if you have a Mac or a PC and if you are running Linux. The following will provide necessary materials to use this book as well as links and some relevant instructions *for those who are using a Mac with macOS.*

A Text Editor. In order to edit files as demonstrated in this book, you will need a text editor outside of the basic default editor. Any editor will work such as Sublime Text[11].

If you are already familiar with an editor or if you like a particular one, use it! However, I personally prefer Visual Studio Code[12] which can be downloaded at the previous link. Throughout this book, I will refer to VS Code with some guidelines on how to find certain features. Should you decide to use another editor, it is up to you to learn how to use it. If you are a beginner with Git and the command line, I would stick with VS Code although as you gain experience you might think about switching to Sublime Text.

Git. To use Git, you need to download and install it on your computer. You already have a command line, you just need to make sure that you have Git. Depending on your computer and where you got it, you may actually already have Git (for instance, university laptops). First, run a check to see if it is already installed. Don't worry about how or why these steps work, all will be explained later on.

1. Press `Command` + `Space` to open Spotlight Search.

2. Start typing `terminal.app`. When the correct application opens up, click on it.

3. Your terminal will pop up. Don't be scared of it! Type `git --version` and press `Enter`.

4. If it spits out something like `git version 2.35.1` (the numbers could be different), you're good! If not, carry on.

Even if you do not know what Homebrew is, it recommend using it to install Git. It can also be used to quickly (i.e. in one line) install other pieces of software. You might find Homebrew useful in the future, even if it can be confusing to install at present:

1. To install Homebrew, go back to your terminal that you opened when checking if Git was installed. Paste in

---

[11]`sublimetext.com/download`
[12]`code.visualstudio.com/Download`

```
/bin/bash -c "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)"
```

as one line (not two) and press Enter. Don't worry about what each part does (to be honest, I'm not entirely sure either). It is safe and trustworthy (I did it on my computer).

If you want to be sure you're pasting the correct command, you can visit the Homebrew homepage[13] and navigate to the "Install Homebrew" heading.

2. Follow the instructions and prompts given until Homebrew is installed.

3. To install Git, simply type in `brew install git` at the terminal and press Enter.

And that's it!

GitHub. You will also need a GitHub account. Since this is an extensive topic, this is covered in Subsection 3.1.1, p. 27.

---

[13]`brew.sh`

Materials: Linux

As this book was written on the assumption that you have no experience with Git or the command line, it also assumes you do not have necessary materials (software). Downloads will differ slightly depending on it you have a Mac or a PC and if you are running Linux. The following will provide necessary materials to use this book as well as links and some relevant instructions *for those who are using Linux.*

Please note that I have no experience with using Linux. These instructions will likely be superficial and without detail. I simply provide you with links; I am currently not able to provide information on how to install software correctly.

A Text Editor. In order to edit files as demonstrated in this book, you will need a text editor outside of the basic default editor. Any editor will work such as Sublime Text[14] which has a Linux download.

If you are already familiar with an editor or if you like a particular one, use it! However, I personally prefer Visual Studio Code[15] which can be downloaded at the previous link (there is a Linux download). Throughout this book, I will refer to VS Code with some guidelines on how to find certain features. Should you decide to use another editor, it is up to you to learn how to use it. If you are a beginner with Git and the command line, I would stick with VS Code although as you gain experience you might think about switching to Sublime Text.

Git. To use Git, you need to download and install it on your computer. You already have a command line, you just need to make sure that you have Git. Depending on your computer and where you got it, you may actually already have Git (for instance, university laptops). You can install Git for Linux from the many options at the GitHub download page[16].

GitHub. You will also need a GitHub account. Since this is an extensive topic, this is covered in Subsection 3.1.1, p. 27.

---

[14]sublimetext.com/download
[15]code.visualstudio.com/Download
[16]git-scm.com/download/linux

Contents

I Introductory Information

II Working Solo

III Working With Others

Appendices

Back Matter

Part I

Introductory Information

Chapter 1

Computer Basics

For those who do not have a lot of experience with computers, start here for an introduction into certain computer basics. This book assumes that you are familiar with the content in this chapter so take some time to browse it. Don't worry, I don't assume you know anything about the "command line" (this will be introduced in Section 2.1, p. 15).

However, for the content of this book to make sense, a few basics must be covered, such as proper file naming and a knowledge of common file extensions. Taking the time to read this chapter *will* pay off in the long run and will make using the command line and Git simpler (and we love simplicity, espeically with computers!).

1.1 File and Folder Names

Take a minute and look at some of the files and folders on your computer. What do the names look like?

Sorry, that probably wasn't very clear. Specifically, I want you to see if there are *spaces* in your file and folder names. My guess is probably! Mine certainly had spaces before I learned more about computers and got involved in Git and the command line.

Take another look at some files/folders. What is your "capitalization scheme"? Is there a pattern? Do you name your files by what they contain? Do you have dates in your files? Do you have a group of files with something like "File First Draft", "File Second Draft", "File, revisions from colleague", etc.? All of these can be problematic for computers and even more so for humans interacting with computers.

The following chunks go through my opinions on file naming conventions. But that's what they are: opinions. Suggestions. Following them will make your life easier later on. But by all means, go ahead and ignore me and decide for yourself later if you want to make a change. I probably won't pop up in your home and check your file names. Try not to get too lost with these; an extensive example will be provided at the end of the section.

1.1.1 Naming Conventions

File Names With Revisions. You might be familiar with this situation: You write one draft, get feedback, save a second draft as a new file, get feedback again, and finally make a new file with a final draft. Then, you decide to make modifications to that final draft which is saved as a new file, etc., etc.This is great until you realize that now you have way too many files for a single document. How do you know which one is real final draft? What if

you open the wrong one and start editing? Now you have to edit all of the files to make them match.

Sure, a computer can handle that many files and doesn't really care how many drafts or revisions you have. But more files take up more storage space and increases the chances of confusing you! How on earth can we solve this problem? It's not like there's a magic software that can keep track of all of our changes in a single file.

Ah, but there is! This magic software is called Git. Git specializes in **version control** which means that you do not have to worry about twelve drafts. Git keeps track of your revision history for you. If you are familiar with Google Docs, Git is very similar. Docs saves your changes and makes it easy to revert back to an old version if you change your mind on anything.

For now, don't worry about Git; we will get there in Chapter 4, p. 45. Just know that after reading this book, you won't have to worry about excess drafts and should never have to save so many drafts again.

File Names With Dates.   Ok, this isn't entirely problematic. But I never understood why dates in file names had to be so specific. Having years makes sense, maybe months. But days? Times? Seconds? Once you start getting that specific with dates and times, you run into the same issue as in File Names With Revisions, p. 3. Dates and times just make your file names incredibly long, hard to read, and hard to use in the command line. Please try to avoid dates, other than years, perhaps.

Descriptive File Names.   File names should be descriptive! Furthermore, they should describe *what the file actually contains/is about* and not some random name. Probably most of your files are descriptive? That's great! But what's the catch? File names should also be short. Short, sweet, to the point, and also descriptive.

That's pretty tricky, you might say. Sure, I respond, but you can take advantage of abbreviations and word-shortening tricks. For instance, suppose I had a file name like this:

Chapters3_4_Overview of Sampling and Simple Random Sampling (SRS).pdf

This is clearly descriptive. We know exactly what this PDF is about: it refers to chapters 3 and 4 of a textbook which is about sampling in general and digs into simple random sampling. But wow, it's really long. On my computer, it shows up as

Chapters3_4_Overview of...SRS).pdf

Now do you know what this file contains? I don't. Yes, it's on chapters 3 and 4 but we are left hanging as to what the chapters are an overview of! Moreover, what does the random "SRS" mean at the end?

If it were me, I would rather name this file (based on the conventions discussed so far)

Ch3_4_Sampling and SRS.pdf

Wow! So much shorter. Do you still have an idea of what the file contains? Hopefully! "Ch" is usually accepted as an abbreviation for "Chapter" so it is clear that this file is over chapters 3 and 4. The subject of these chapters is sampling and simple random sampling ("SRS" is a widely-used abbreviation in statistics for simple random sampling).

File Naming Trick.

An easy way to shorten file names is to remove the vowels from words. The human brain is incredible in that it can determine what a word should be based on a small amount of context. Assuming the file above was in a folder pertaining to statistics, I could shorten it even more:

     Ch3_4_Smplng and SRS.pdf

And we still know what it's about!

    A quick word of caution. Take care not to over-abbreviate. It may not be necessary to abbreviate every word. For instance, I wouldn't change "you" to "y" or "gate" to "gt". You may be confused later: does "gt" mean "gate", "git", "get", "gut", "agate", etc., etc.?

Summary: keep your file names short and to the point. Describe what the file is (I wouldn't name the file above "bananas.pdf"!).

File Names With Spaces. If you only follow one convention in this section, this would be the one. This the most important for working with the command line. Spaces in file names increase the amount of typing you have to do and make it more difficult to understand what's going on on the command line.

For instance, suppose I wanted to open the file we discussed earlier. On the command line, I would open the file like this (don't worry about the commands, just notice what it looks like):

```
open 'Chapters3_4_Overview of Sampling and Simple Random Sampling (SRS).pdf'
```

I could also do it like this:

```
open Chapters3_4_Overview\ of\ Sampling\ and\ Simple\ Random\ Sampling\
(SRS).pdf
```

First, notice how long these titles are. That's a lot to type! What you should really notice is that in the first example, the file is surrounded in quotes and in the second, there are a bunch of backslashes. For each file name with spaces, you have to either remember to enclose the entire name in quotes or to take the time to put a backslash before every space. It may not seem like too much of a hassle, but you will get annoyed with it pretty quickly. It also can cause setbacks inside code editors when you want to import or export files with spaces.

So what do we replace the spaces with?

Note 1.1.1.1 Alternative Naming Options. Most computer programmers name their files using **cases**. There are five main cases:

snake_case	no capital letters, spaces are replaced with underscores (_)
kebab-case	no capital letters, spaces are replaced with hyphens (-)
camelCase	first word is lowercase, all consequent words are capitalized; spaces are removed
PascalCase	every word is capitalized, spaces are removed
UPPER_CASE_ SNAKE_CASE	every letter is capitalized, spaces are replaced with underscores (_)

See a Most Common Programming Case Types blog post[17] to get more details on these cases.

Personally, I use snake_case for file names and kebab-case for folder names, just to help me keep them separate and still easy to read. I also tend to name my files and folders with lowercase letters if I can.

Using the shortened file name above, I would use the following as my file name:

ch3_4_smplng_srs.pdf

Note that I got rid of the "and". Cases are useful in that they make it easy to remove articles and prepositions which in turn helps us keep file names nice and short.

It is also important to keep your file names consistent. If I had another file about chapter 5 of this textbook which is about cluster sampling, let's say, I would want to name it something like

ch5_clstr_smplng.pdf

Notice that the structure is the same as before: first I have "ch5" to represent the chapter number, then I describe the chapter with "clstr_smplng". Like before, I use snake_case and I made sure to use the same abbreviation for "sampling" as I did before. Consistency is key for our own sanity and so we can quickly scan for the file we need.

It's worth repeating once again. File names should not contain spaces. Files about related content should have a similar naming scheme.

Sorry, once more. This time, I'm going to yell. ***File names should not contain spaces.***

A Quick Note on Folder Names. Folder names should also follow all of the conventions above. This is a lot harder to get used to. I still feel like I'm breaking the Human Code of Folder Names everytime I ignore spaces and capital letters. For whatever reason, I'm fine with file names, but folder names just don't feel right.

But too bad for me! I name my folders according to the conventions anyways. And you should too. (As I mentioned earlier, I use snake_case for files and kebab-case for folders. You might consider doing so as well; they are the two most popular cases.)

1.1.2 Exploring File and Folder Naming Techniques

There were a lot of words in Subsection 1.1.1, p. 3. Let's make sense of them here with an extensive example. There will be some questions throughout the way. I encourage you to think about them and make sure you have understood the naming conventions above.

Activity 1.1 Folder Naming. Suppose you work for Taylor Swift (wouldn't that be cool!). She is super busy right now recording albums, writing songs, spending time with family and friends, and making TikToks and needs your help. She has all of her songs saved on her computer in folders which are divided up by one album per folder. She has asked you to help her rename her folders so that she still knows what each folder is, but follows proper naming conventions. Use your knowledge from above to help Taylor out.

Following is a list of her current folder names:

---

[17]chaseadams.io/posts/most-common-programming-case-types/

- Taylor Swift
- Taylor Swift (Deluxe)
- Live From Clear Channel Stripped 2008
- Fearless
- Fearless (Platinum Edition)
- The Taylor Swift Holiday Collection
- Speak Now
- Speak Now (Deluxe Edition)
- Speak Now World Tour Live
- Red
- Red (Deluxe Edition)
- 1989
- 1989 (Deluxe Edition)
- reputation
- Lover
- folklore
- folklore (deluxe edtion)
- folklore: the long pond studio sessions (from the Disney+ special) [deluxe edition]
- evermore
- evermore (deluxe version)
- Fearless (Taylor's Version)
- Red (Taylor's Version)
- Midnights
- Unreleased Music
- Singles

Wow! That's a lot of albums! Complete the following tasks to make Taylor's life a little easier for the future.

Please note that the answers to many of these exercises will vary depending on personal preferences. I will give answers depending on how I might approach the problem; they are certainly not the only answer you could have given.

(a) Before we trying fixing a problem, let's figure out what the problem is. What do you notice about these folder names? What about them is "incorrect"?

(b) Ok, we've identified a problem. How can we go about deciding what to do about it? You may already have some ideas, but stick with me here. Let's not just dive in and start removing spaces and shortening folder names. Instead, let's make a plan.

Identify some naming patterns. Are there any album names that are similar? Can you form any groups of names?

(c) I am going to continue with the groups in the solution to Task 1.1.b, p. 7. Feel free to branch off with your own groups or stick with me.

Now that we have groups, we can figure out how to name one of each group, then apply that naming style to all names in the group. Start with the standard albums, such as "Taylor Swift". How can you change this name to stick with the conventions discussed above?

(d) Apply the naming to scheme to all folders with "regular" album titles.

(e) Continue with the deluxe group. Identify patterns, change one name, then apply those changes to all the names in the group. Use your best judgement, but don't stress about the "perfect" name.

(f) Continue with the live group. Identify patterns, change one name, then apply those changes to all the names in the group. Use your best judgement, but don't stress about the "perfect" name.

(g) Continue with the Taylor's Version group. Identify patterns, change one name, then apply those changes to all the names in the group. Use your best judgement, but don't stress about the "perfect" name.

(h) Continue with the other group. Identify patterns, change one name, then apply those changes to all the names in the group. Use your best judgement, but don't stress about the "perfect" name.

I think that's all of them! That may have seemed like a lot of work but I hope it was worthwhile. It should now be much easier for Taylor to navigate her folders and quickly know that album they contain.

If you feel like you have a good grasp of naming systems, feel free to skip Activity 1.2, p. 8. If not, let's explore file naming a little further. In Activity 1.1, p. 6, we focused on Taylor's folder names and since you helped her there, she is asking for your guidance with her file names as well.

Activity 1.2 File Naming. We will focus on the folder that I renamed to be `speak-now-dlx`. A list of her current file names for these tracks follow. I should note that it is often helpful to prefix song names with their track position so that they stay in order when in a folder. We won't remove those numbers. Also, ".wav" indicates that the song is a WAV file (see Section 1.2, p. 9 for more). We should not remove these either as this could result in file loss or corruption.

- 01 Mine.wav
- 02 Sparks Fly.wav
- 03 Back To December.wav
- 04 Speak Now.wav
- 05 Dear John.wav
- 06 Mean.wav
- 07 The Story Of Us.wav
- 08 Never Grow Up.wav
- 09 Enchanted.wav
- 10 Better Than Revenge.wav
- 11 Innocent.wav
- 12 Haunted.wav
- 13 Last Kiss.wav
- 14 Long Live.wav
- 15 Ours.wav
- 16 If This Was A Movie.wav
- 17 Superman.wav
- 18 Back To December (Acoustic).wav
- 19 Haunted (Acoustic Version).wav
- 20 Mine (POP Mix).wav

Like before, complete the following tasks to make Taylor's life a little easier for the future.

Again note that the answers to many of these exercises will vary depending on personal preferences. I will give answers depending on how I might approach the problem; these are not the only correct responses.

(a) Before we trying fixing a problem, let's figure out what the problem is. What do you notice about these file names? What about them is "incorrect"?

(b) Ok, we've identified a problem. How can we go about deciding what to do about it? You may already have some ideas, but stick with me here. Let's not just dive in and start removing spaces and shortening file names. Instead, let's make a plan.

Identify some naming patterns. Are there any track names that are similar? Can you form any groups of names?

(c) I am going to continue with the groups in the solution to Task 1.2.b, p. 8. Feel free to branch off with your own groups or stick with me.

Now that we have groups, we can figure out how to name one of each group, then apply that naming style to all names in the group. Start with the "regular" tracks. How can you change these names to stick with the conventions discussed above?

(d) Apply the naming to scheme to all the acoustic tracks.

(e) Apply the naming to scheme to all the last track.

And that's it! If you would like more practice, look up the track listing for her other albums and repeat this activity. It would be a great activity in consistency; for example, for the tracks on the Taylor's Version albums, you would want to use the same naming scheme as in the folders. For me, I would say `02_red_tv.wav` for "Red (Taylor's Version)" and to any acoustic tracks, I would append `_acoust`.

1.2 File Extensions

There are many different types of files. At the very basics, we have images, videos, documents, and songs, but there are so many more. Even within each of those categories, there are numerous types. You may be familiar with a PDF and a Word Document. Both of those could be classified as "documents", but are interacted with very differently. Whether you are an advanced computer programmer or someone who uses a computer for every-day purposes, an understanding of file extensions is essential. **File extensions** help us understand what type of file we are working with and give us an idea of how we can interact with that file. The following chunks give common (and some not so common) extensions for various categories. Refer back to this section often, espeically if you find yourself forgetting what a file is.

Hint: `Control` + `F` or `Command` + `F` work well on this page.

1.2.1 Images

Why would we need more than on extension for an image? A photo is a photo, right? Wrong. Each of the file extensions have unique aspects and features. For instance, the Apple `.heic` file often takes up less storage space. However, not all programs can open them. The University of Michigan[18] and Kinsta[19] have great guides on the differences between some image file extensions. The table here just lists the extensions and what they are, not the details on their use.

---

[18]`guides.lib.umich.edu/c.php?g=282942&p=1885348`
[19]`kinsta.com/blog/image-file-types/`

Table 1.2.1.1 Image File Extensions

Extension	File Type
.jpg or .jpeg	Joint Photographic Experts Groups image
.png	Portable Network Graphics image
.svg	Scalable Vector Graphics image
.gif	A looped image/video blend
.eps	Encapsulated PostScript (vector) image
.bmp	Bitmap image
.tif or .tiff	Tagged Image File Format
.raw	Raw image (usually from a camera)
.HEIF	High Efficiency Image File
.HEIC	High Efficiency Image Container (Apple)
.psd	Adobe Photoshop
.ai	Adobe Illustrator

1.2.2 Videos

Adobe[20] has a great guide on the differences between some video file extensions. The table here just lists the extensions and what they are, not the details on their use.

Table 1.2.2.1 Video File Extensions

Extension	File Type
.mp4	MPEG-4
.mov	QuickTime Movie
.wmv	Windows Media Viewer
.avi	Audio Video Interleave
.prproj	Adobe Premiere Pro Project
.aep	Adobe After Effect Project
.fcp	Apple Final Cut Project
.camproj	Camtasia Project
.m4v	iTunes Movie File

1.2.3 Documents

There are many different file extensions that refer to documents. Many of them you will never encounter. Only the main file types are listed here. File Stack[21] has a good explanation on some of the file types here.

---

[20]adobe.com/creativecloud/video/discover/best-video-format.html
[21]blog.filestack.com/thoughts-and-knowledge/document-file-extensions-list/

Extension	File Type
`.doc` or `.docx`	Microsoft Word Document
`.xsl` or `.xslx`	Microsoft Excel Spreadsheet
`.ppt` or `.pptx`	Microsoft PowerPoint Presentation
`.pages`	Apple Document
`.key`	Apple Keynote Presentation
`.numbers`	Apple Numbers Spreadsheet
`.pdf`	Adobe Portable Document Format
`.rtf`	Rich Text Format Document
`.txt`	Plain Text Document
`.log`	Text Log File (often records messages)
`.md`	MarkDown Document
`.Rmd`	MarkDown Document for use in R

Note: technically, `.html` and `.xml` are considered document-type files. I am choosing to place them under coding-type files instead.

1.2.4 Audio

makeuseof.com[22] has great descriptions of the most common audio file types.

Extension	File Type
`.mp3`	MPEG Layer 3 (lossy compression)
`.wav`	Waveform Audio Format (uncompressed)
`.aiff`	Audio Interchange File Format (uncompressed)
`.aac`	Advanced Audio Coding (lossy compression)
`.wma`	Windows Media Audio (lossy compression)
`.flac`	Free Lossless Audio Codec (lossless compression)
`.aa` or `.aax`	Audible Audio File
`.m4a`	Audio-Only MPEG-4 (iTunes)
`.midi`	MIDI File

1.2.5 Coding and Programming

There is a plethora of programming languages, software, and interfaces and each of them uses a different file extension.

---

[22]`makeuseof.com/tag/audio-file-format-right-needs/`

Extension	File Type
.R	R Script
.py	Python Script
.ipynb	Jupyter Notebook
.html	HyperText Markup Language
.css	Cascading Style Sheets
.js	Javascript
.sas	SAS Script
.xml	eXtensible Markup Language
.xslt	eXtensible Style Sheet Transformations
.ptx	PreTeXt Markup File
.tex	LaTeX Document (typically for mathematics)
.c	C and C++ File
.cpp	C++ File
.java	Java Source
.swift	Swift Code File
.vb	Visual Basic File
.vbs	Visual Basic Script

1.2.6 Data

Many of these extensions also belong in Subsection 1.2.3, p. 10 but are related enough to merit another subsection.

Extension	File Type
.csv	Comma Separated Values
.tsv or .tab	Tab Separated Values
.json	JavaScript Object Notation
.csv	Comma Separated Values
.tsv	Tab Separated Values
.rds	Single R Object
.RData	Multiple R Objects
.sas7bdat	SAS Data File

1.2.7 Other File Exensions

Extension	File Type
.band	Apple GarageBand File
.logicx	Apple Logic Pro File
.exe	Windows Executable File
.zip	Compressed Files
.tar.gz	Collection of (archived) files
.app	Mac Application
.dmg	Mac Disc Image (executable file)
.pkg	Package of Software or Files
.otf	Open Type Font
.ttf	TrueType Font

1.3 File Paths

Imagine that I bring you to the edge of a forest, an immense, thick, unexplored forest. When we get there, I tell you that somewhere in that forest lays a chest of pure diamonds, put

there 1000 years ago. Then I give you a task: I need you to go find that chest for me (yes, you get a portion of the loot).

Think about how you might go about this task. The forest is unexplored; we only know that the chest exists inside. Where will you look first? Where will you enter? How will you keep track of where you've been? If you find it, how will you get back? There are many issues preventing you from doing your task efficiently. Sure, you could get lucky and the chest might be only 50 feet from the edge. But maybe it's 3000 feet away (or more!).

Think of your computer like the forest. If I ask my computer to open a file called `poem.pdf`, how will it know where to look for that file? Like you in the forest, it wouldn't even know where to start. Likewise, what if I had two files named `poem.pdf` in different folders? What if I had two files named `poem.pdf` in the same folder? How would it know which one to grab? It wouldn't! It's like me telling you there are two identical chests of diamonds in the forest and I want you to bring me the one I am thinking about. But you don't know the one I am thinking about.

This situation necessitates file paths. Literally, they are the direct path through which the computer can find the exact file you are thinking of. You are basically telling the computer where the file is. Continuing the forest example, it's as if I told you to enter the forest, take a right at the first tree with a triangle, turn left at the lake, and walk six steps past the boulder that looks like a watermelon. You'd know exactly how to get there and would always find the correct file.

1.3.1 Constructing File Paths

What do computer file paths look like? You might have seen one before. Here is an example of a file path on a Mac:

```
/Users/<username>/Documents/poem.pdf
```

With this you can see exactly where my poem lies. First, go to a list of all the users on my computer. Second, look at the files for my username, go to the documents folder, and access the file called `poem.pdf`. Easy!

Maybe. Here's what that would look like on a Windows computer:

```
C:\Users\<username>\Documents\poem.pdf
```

That looks a little different. This one says to go to the C drive. Then look at the list of the users on the computer. Then go to the files for my username, then the documents folder, then the file `poem.pdf`.

However, they both work the same. The fundamental structure is "start at the outmost folder and work your way in through folders until you get to the file or folder you are looking for". Recall from Section 1.1, p. 3 that files must have an extension while folders do not. This is an easy way to differentiate between folders and files.

The key thing to note is that Mac and Windows are designed differently. Mac uses forward slashes to indicate a folder change and Windows uses backslashes (in Git Bash, Windows users can use forward slashes…one reason for sticking with Git Bash!). But their structures are different as well. Notice that Windows has "C:" at the beginning. These are called "drives"; this is the C drive here. There are other drives based on what is stored where but this and other differences are beyond the scope of this book. A brief history on file paths is given by How-To Geek[23] and a rather hard-to-read explanation can be found on Wikipedia[24].

---

[23]howtogeek.com/181774/why-windows-uses-backslashes-and-everything-else-uses-forward-slashes/
[24]en.wikipedia.org/wiki/Path_(computing)

There are a few important points to remember when reading or finding a file path. The main goal from this section is that you understand how a file path is structured. Paths will come back later.

- File paths do not like spaces. Recall the lesson learned in File Names With Spaces, p. 5. To a computer spaces indicate that you are ending the file path and starting a new command. Do not name folders or files you are planning on accessing with the command line with spaces.

- When you want to enter a new folder, use a slash to indicate that you are going further in the document tree.

- File paths are unique; you cannot have one file path refer to two objects. This is why your computer will never let you have two files with the same name and extension in the same folder.

  You can however have files with the same name in different folders. Since the file path is different, there are no issues with duplicate files.

1.3.2 s and Reminders

File and Folder Naming.   Recall Note 1.1.1.1, p. 5 for alternative naming methods for files and folders. Please, please, please do not use spaces (and consider renaming your existing files that you plan to use with the terminal). I know it looks weird to name things using these cases, but if you really are set on learning Git and the command line, you will want to follow this convention.

Finding File Paths.   In theory, you will never need to type out the full file path. Your computer actually records this information for you to just copy.

On a Mac, right click on any file or folder you want the path for. Click "Get Info". Under a label called "Where:" you can find the path. Right click on that path and hit "Copy as pathname" to copy the path in the same format as the paths above.

On Windows, right click on any file or folder you want the path for. Click "Properties". Under a label called "Location:" you can find the path. Highlight the text and copy it.

Using the fish Shell.   *Advanced users only.*

If you changed your shell to fish as described in Appendix A, p. 83, you won't even have to copy a path or remember a path. fish contains autocomplete that looks for files and folders in your current folder and recommends completions.

Chapter 2

The Command Line

So you came here to learn about Git?

Stop right there!

In order to learn Git, let's begin with something a little less complex: the command line. Yep, the "hacker screen", the plain-looking interface you often see in spy movies. While we won't be doing any hacking, we will need to have and understanding of the command line, its purpose, and some of its useful functions.

Beware to all beginners who skip this chapter! The most popular way to use Git is through the command line; without at least a basic understanding of it, you will have a hard time with the later subjects.

2.1 What is the Command Line?

The first step to tackling the command line is to not be scared. Sure, it looks daunting and if you make a mistake you might produce some lengthy error messages, but at the very minimum, the command line *is just another way to interact with your computer*.

The **command line** (also known as a **command prompt**, **command-line interface**, or **terminal**) is an application that accepts lines of text and sends that text to your computer which performs an action.

Terminals are powerful and fast. Think of them as a more direct way to interact with your computer. You can delete files with the terminal. You can create new files. You can move files. You can open files. And more. There are many, many complicated and hard-to-comprehend functions of the command line. Rest assured, this chapter sticks with the basics; we will learn necessary functions for using Git (and a few others).

You may also hear about shells. A **shell** helps make the command line more interactive and user friendly. Some shells add colors, informative information, and/or predictive text. We will return to shells and command line customization in Appendix A, p. 83. For now, do not worry about the details behind shells. A knowledge of them is not necessary to use Git, but they can help make things a little easier.

2.2 Where is the Command Line?

If you have a computer, you have a terminal. You may not have ever used it or known it was there, but it's hiding, waiting for use. Let's find it.

2.2.1 On a Mac

Luckily, the terminal is easy to find on a Mac: it's called `terminal.app` and can be found in numerous ways.

- (Recommended) Press ⸢Command⸥ + ⸢Space⸥ to open Spotlight Search. Start typing "terminal" and the app should pop up quickly. Press ⸢Enter⸥ or click on the result.

- Navigate to the Applications folder in the Finder. Then, open the Utilities folder and double click on `terminal.app`

- Find it through the Launchpad. What's the Launchpad, you ask? On most Mac keyboards, it is on the key corresponding to ⸢F4⸥ and looks like a bunch of boxes arranged in rows. You can also access the Launchpad by clicking the Launchpad icon in the Dock (also boxes arranged in rows) or by going into the Applications folder in the Finder and double clicking on the `Launchpad.app` icon.

  Once in the Launchpad, find the Terminal. You can either search for it in the search bar at the tope of the screen or scroll through your apps until you find it. (Note: it might be in the Other folder).

Shells.

At the top of my (Mac) terminal, it says "<my-username>— -bash". This indicates that I am using the `bash` shell. This is no longer the default for MacOS Catalina and later (as the terminal states). I will be sticking with `bash` in this book to avoid major differences between Windows and Mac instructions. If your terminal states something other than `bash` (such as `zsh` or `fish`) and you would like to use `bash` to follow along, paste `chsh -s /bin/bash`[25] into the command line, press ⸢Enter⸥, enter your password to your computer, and restart the terminal. Otherwise, you are free to stick with the default shell; just know that there may be slight differences between this book and your terminal.

2.2.2 On Windows

For Windows users, the situation is a slightly more complicated. The underlying makeup of the operating systems on Windows is differet than the basis for MacOS and other Unix-like systems (such as Linux). This means that certain applications for Windows do not have a MacOS compliment (and likewise some Mac applications do not have Windows versions). In addition, the file system and organization are different bewteen the two.

Separate instructions must be given for Windows users for certain tasks. To make life easier for you, me, and everyone else, I recommend *not* using the default Windows terminal.

The default terminal for Windows is usually the Command Prompt. To access it, activate the search feature of your computer by clicking the search box at the bottom of your screen or by typing ⸢Windows⸥ + ⸢S⸥. Then search for "command prompt" and click on the result that comes up.

Instead, let's use a new terminal (Git Bash) that works with Git and conveniently uses the `bash` shell which allows me to use the same instructions for Mac as Windows. Detailed download instructions can be found in Materials: Windows, p. xiii. I will assume that you are using Git Bash for the remainder of the book.

---

[25]This command stands for "CHange the SHell to the shell called bash in the bin folder"

2.2.3 On Linux

This will depend on the version and distribution you have installed. For this reason, I will not be able to provide instructions here. Please use the internet for help. Contributions for your specific distribution are welcome through a pull request or issue on GitHub.

2.3 Basic Commands

Now that we have understand what a terminal does and how to find it, let's learn some commands. We'll start with the basics.

2.3.1 Navigating Your Computer With the Command Line

When your first open your terminal, you aren't given very much information. On Mac and Windows, you get your username and maybe your computer name/id but not much else. This isn't very helpful. How can we do anything with our computer when we don't even know where we are on the computer?

Using `pwd`. Our first command will show us where we are: `pwd`, which stands for **present working directory**.

Note 2.3.1.1 Folders Are Not Directories. Until now, I have been using "folder" when I probably should be using "directory". From Wikipedia[26] we can read that

> If one is referring to a container of documents, the term folder is more appropriate. The term directory refers to the way a structured list of document files and folders are stored on the computer.

That's a little confusing! The difference here is mostly technical. Essentially, a **directory** is a "file system object" whereas a **folder** is just a user-friendly way to represent directories. For our purposes (and for most purposes), the two terms can be used interchangeably. I use "directory" when speaking about functions on the command line and "folder" when speaking in informal settings. If you are interested, you can read more about the history of the terms at this Quora question[27].

Checkpoint 2.3.1.2 Trying `pwd`. Enter `pwd` at your terminal. What does it tell you?

Helpful Hint.

Anytime I talk about "entering something at your terminal", type in the desired command and press `Enter`.

`pwd` prints one thing: the file path to the folder that is "active" at your terminal. When you first open your terminal, Mac users might see something like `/Users/<your-username>` and Windows (Git Bash) users `/c/Users/<your-username>`. If you ever get lost in your terminal, type in `pwd` for some reassurance.

---

[26]en.wikipedia.org/wiki/Directory_%28computing%29#Folder_metaphor
[27]quora.com/What-is-the-difference-between-a-folder-and-a-directory

Using cd.   Knowing where you are is great, but as of right now, we are stuck in one place. What use is pwd if we can't move anywhere? By default, your terminal places you in the "highest" level possible for the active user; from there we can only move "in". Essentially, you are moving from one directory *into* another. The cd, or **change directory** command can do this for us. cd requires another piece of information: the directory you wish to navigate to, which must be typed using file-path syntax.

File-path syntax refers to how we describe where a file lies on your computer. The basic structure is as follows:

`/Users/your-username/main-folder/sub-folder1/.../file_name.file_extension`

Note the forward slashes; the slashes separate folder names whereas periods separate file names and file extensions.

Checkpoint 2.3.1.3 Trying cd. Enter `cd Desktop/` into your terminal. Did anything change?

Hint.   Try running pwd after cd. Do you notice anything different here?

If everything went correctly, you should now be "in" your Desktop directory (folder, if you insist). You may have even noticed that your terminal changed. On any operating system, you should see that the word "Desktop" appeared before your cursor. This is a nice check to make sure that cd worked and that you are where you want to be.

Using ls.   Ok, we're making progress! Now that we are in the Desktop, what can we do? Let's use another terminal command to see what files and directories are currently on your Desktop. ls, or **list contents** will do the trick. Note that the output will differ for each user as we all have different files in different places.

Checkpoint 2.3.1.4 Trying ls. Enter `ls` into your terminal. What is the output? Is it what you expected?

Perhaps unsurprisingly, your terminal should have just listed every single element on your Desktop. If you every forget what files are in your "active" directory (and you don't want to open your file browswer and navigate to the folder), use ls as a refresher.

You may have noticed that some files/directories show up in your terminal that are not visible on your Desktop. This is because most operating systems by default hide certain files that should not be deleted. They still exist, they are just invisible to prevent accidental deletion.

2.3.2 Adding Files and Directories

Now that we are familiar with navigating our computer with the command line, let's learn a little more. Suppose you are in your Desktop (which we are) and you want to add a text file. Since you are already in the terminal, you don't want to open a text editor, create a new file, and save it to your desktop. Or, suppose you have a file on the Desktop that you no longer need and you don't to open you file browser, navigate to the Desktop, and delete the file. These examples may seem silly, but now suppose you are six folders deep. It would take much more time to navigate to those folders in a file browser than it would in the terminal.

Using touch.   Creating a new file is simple. The command requires three parts:

1. The word touch,

2. The file name, and

3. The file extension.

All **touch** does is create an empty file with the name and file extension specified by you on the command line. Why touch you ask? Well, everytime you edit ("write to") the file, you are "touching" it electronically.

Note: Don't forget to follow Note 1.1.1.1, p. 5 when creating your files!

Checkpoint 2.3.2.1 Trying touch.

(a) Enter touch test.txt into your terminal. What happened? Did it work as you expected?

(b) Can you use a command we learned earlier to check if your touch worked?

Oh dear, nothing happened! When you press enter after a touch command, the terminal gives you no output at all. How can we verify that a test.txt file got created? Let's try ls. Recall from Using ls, p. 18 that we can use ls to give us a list of all the items in our present working directory. (Remember that term? See Using pwd, p. 17.) Since we are currently on the Desktop and we want to see what files are on the Desktop, we can use ls to see if our file was created. Try it now!

Activity 2.1 File Extensions Matter. When touching a file, you *must* put the file extension at the end of the file name. Otherwise, your computer won't know what type of file to create!

(a) Enter touch text at your terminal (it's safe, don't worry).

(b) Everything should still work as before. Did the terminal print any output? Try running ls. Does a file named "text" appear?

(c) Find the text file in your file browser and try to open it (by double clicking.) What happens?

The purpose of this activity is to demonstrate the importance of file extensions. When none are provided, your computer either has to guess what the file's contents are or simply cannot interpret the file and asks you for help. We would like to avoid computer confusion as much as possible. Common file extensions and their file types are listed in Section 1.2, p. 9.

Just remember, anytime you wish to create a file, you need touch, a name, and an extension.

Using open. Once a file is created, opening it is simple: just use the open (Mac) or start (Windows) command. To **open** or **start** a file, we use a syntax similar to touch:

1. The open or start command,

2. The file name, and

3. The file extension.

Your terminal will choose the correct software to open the file based on the file extension, once again stressing the importance of Activity 2.1, p. 19. Changing default programs is not a part of this book, but a Google search should do the trick if you are interested.

Checkpoint 2.3.2.2 Trying `open`. In Checkpoint 2.3.2.1, p. 19, we created a new file called `test.txt`. Open this file with the terminal. If you are on a Mac, don't close it yet! Windows users may close the file manually.

Using `killall`. *This command is only on Mac.*

Closing a file is less trivial. It can be tricky to close a *single* file, but closing an *entire application* is not too bad. You need two pieces:

- The command `killall`

- The name of the application you wish to close.

A few remarks: The **kill all** command will quit the entire application, not just a single file. Be sure you want to quit (i.e., you have saved any changes) before using kill all. Also, application names are case sensitive; that is, `killall TextEdit` will work fine (on a Mac), but `killall textedit` will return an error. Finally, be aware that if an application name has a space in it (e.g. Microsoft Word), you will need to enclose that name in quotes: `killall 'Microsoft Word'`. You can use single or double quotes as long as you use the same style for both.

Checkpoint 2.3.2.3 Trying `killall`. Use the terminal to close the `test.txt` file that we just opened in Checkpoint 2.3.2.2, p. 20. Remember that the command is case sensitive!

Using `mkdir`. We've seen how to use the terminal to create new files; however, sometimes files aren't enough. With the command line, we can also create new directories (represented by folders) for us with `mkdir`. The **make directory** command does exactly that: makes a new directory.

Checkpoint 2.3.2.4 Trying `mkdir`.

(a) Enter `mkdir testdir` on your command line. What happens? Is this what you expected after learning about `touch`?

(b) Use `ls` to verify that the command worked.

You should see, with `ls`, that a new directory was created. Windows users may see that their new "testdir" is a different color (and has a "/"). They both indicate that "testdir" is a directory. Mac users may not see these which thus stresses the importance of Activity 2.1, p. 19. When `ls` is used, the file extension is printed (when there is one). Directories do not have file extensions so when interpreting `ls` output, it can usually be safely assumed that any item without an extension is a directory. Note that for any OS, Note 1.1.1.1, p. 5 applies for `mkdir` as well.

Activity 2.2 Putting It All Together, Part 1. It's time to put your skills to the test! Using your terminal and the knowledge gained from this section so far, complete the following tasks. This activity assumes you have been following along with the checkpoints.

(a) Navigate into the newly-created directory, "testdir".

(b) Determine if there are any files inside of this directory. What is the file path to "testdir"?

(c) Create a new text file with a name of "My Greeting". Be sure to use proper naming techniques and correct terminal syntax.

   Hint. You can use `ls` to verify that everything worked.

(d) Open your text file and type a greeting into the first line. Save the file. Remember to only use the command line to open the file!

(e) Close your text file. Mac users should use the command line whereas Windows users will have to close it manually.

2.3.3 Removing Files and Directories

Over the course of this section, we have added useless files and directories. We do not need our test files or folders anymore, so let's learn how to delete them (with the terminal, of course!)

Using `rm`. **Removing** a file with `rm` has the exact same syntax as using `touch` (Using `touch`, p. 18) except that instead of `touch`, we use `rm`. To summarize, we would need

- The command `rm`,

- The file name, and

- The file extension.

Checkpoint 2.3.3.1 Trying `rm`.

(a) Use `ls` and `pwd` to verify that you are still in the `testdir` directory and that the file `my_greeting.txt` exists.

(b) Remove the file `my_greeting.txt`

(c) Use `ls` to verify that the process was successful.

Warning 2.3.3.2 Removing Files is Permanent. Removing files is easy...Too easy. Notice that the terminal provided no output or verification that the process was happening. Also notice that there was no dialog box asking if we were really sure that we wanted to remove the file.

   Removing files with the terminal is permanent. There is no recycle bin for these files. There is no "undo" or "restore". The terminal deletes and forgets. Use extreme caution when using `rm`; only remove what you are *absolutely sure* you need to remove. You will not get a second chance and your computer will not ask you if you for verification.

`cd` Backwards. Alright! We cleared the contents of our `testdir` directory. Remember, the ultimate goal of this part is to delete *all* of the new files and folders we created in this section. But now we're stuck. The terminal is still in the `testdir` directory (which now contains no files). In order to delete our other files we have to **change directories backwards**. Luckily, we can still use `cd`, but instead of supplying a folder to move into, we give `..` instead. Together, `cd ..` tells the terminal to move to the directory above the one it is currently in (the **parent directory**).

Checkpoint 2.3.3.3 Trying `cd` Backwards. Try it! Move backwards into the parent directory of `testdir`.

Hint.   If successful, `pwd` should indicate that you are in the Desktop.

Using `rmdir`. You have the tools you need to remove files, but what about directories? If you'd like, try removing our `testdir` directory with `rm`. What happens?

You should get an error stating that `rm` cannot remove directories. Fortunately, there is an aptly named command called `rmdir` which helps us **remove directories**. Using `rmdir` is exactly like using `rm` except that you can only type names of directories.

Checkpoint 2.3.3.4 Trying `rmdir`. Use `rmdir` to remove the test directory, `testdir`.

Warning 2.3.3.5 Removing Directories is Permanent. Similar to Warning 2.3.3.2, p. 21, removing directories is too easy. Notice again that the terminal provides no output or verification that the process was happening. Also notice that there was again no dialog box asking if we were really sure that we wanted to remove the directory.

Removing directories with the terminal is permanent. There is no recycle bin for these files. There is no "undo" or "restore". The terminal deletes and forgets. Use extreme caution when using `rmdir`; only remove what you are *absolutely sure* you need to remove. You will not get a second chance and your computer will not ask you for verification.

Note: you can add in an extra layer of security with the `sudo` command, which requires your computer's password to be entered.

Activity 2.3 Putting It All Together, Part 2. You now know the basics of using using the command line. In this activity, you will practice what you have learned. Be sure to use the command line for each task.

(a) Verify that you are still on your Desktop. Remove the rest of the test files we created throughout the chapter. Check to make sure all the files have been removed.

(b) Navigate backwards one folder level.

(c) Navigate to your `Documents` folder. If you don't have one, create one and then navigate to it.

(d) Create a new folder called "My Favorites". Remember to use proper naming techniques.

(e) Navigate into your newly-created directory. Verify that you are there.

(f) Create three files: "Food" (a text file), "Hobbies" (a Word document), and "Smells" (an Excel spreadsheet). Verify that these were created correctly.

(g) Open each of the three files, type your top three favorites of each category, and close the files again. (Remember, Windows users will have to close the files manually).

(h) Sorry, I know you just edited the documents, but now, it's time to delete them. Remove all three of the files you edited. Verify they were removed.

(i) Navigate back to your Documents folder. Verify you are indeed there.

(j) Remove the directory we created in this activity. Verify that is was removed.

2.4 Advanced Commands

If you seek a more advanced glance into the potential of the command line, you have come to the right place. Each of the following commands are not necessary to proceed through this book but may ease your workflow in the future. As you get comfortable with the terminal, consider adopting some of these commands.

Using `sudo`. According to University Information Technology Services[28],

---
[28]`kb.iu.edu/d/amyi`

> The sudo command allows you to run programs with the security privileges of another user (by default, as the superuser). It prompts you for your personal password and confirms your request to execute a command by checking a file, called sudoers, which the system administrator configures. Using the sudoers file, system administrators can give certain users or groups access to some or all commands without those users having to know the root password. It also logs all commands and arguments so there is a record of who used it for what, and when.

This is a little too much for what is necessary here but has good information. Essentially, `sudo` exists to allow you to run commands as an admin who, in some cases, might have more permissions to run certain commands. The benefit of `sudo` is that your password is required.

So, if there is ever a command that involves risky business (such as permanently deleting files or folders) or if you ever want to purposefully add a layer of safety, use the `sudo` command. The command is actually very simple: add the word `sudo` to the beginning of any terminal command and you will be required to enter your computer's password before the command executes.

Using `mv`. The `mv` command has many uses, all revolving around moving and renaming files. There are a few use cases as described below. The title of each case is the situation you may find yourself in followed by a shell of the command, an example command, and a more detailed description of the command. This content is inspired by the University of Alberta[29].

You already have a file and you want to *rename* it

        mv <existing-file-name> <new-file-name>

        mv birds.txt cats.txt
        Takes prexisting file birds.txt and *renames* it to cats.txt

You already have a file and you want to *move* it AND *rename* the file

        mv <existing-file-name>
                <destination-folder-name>/<new-file-name>

        mv birds.txt animals/cats.txt
        Takes prexisting file birds.txt, *renames* it to cats.txt and *moves* it to the animals/ folder. (Also removes the original birds.txt)

You already have a file and you want to *move* it AND keep the same file name

        mv <existing-file-name> <destination-folder-name>

        mv birds.txt animals/
        Takes prexisting file birds.txt and *moves* it to the animals/ folder.

You already have a folder and you want to *rename* it OR you want to *move* files and folders in an existing folder to a different folder

        mv <folder-to-move> <destination-folder-name>

        mv drums/ instruments/
        Takes prexisting folder drums/ and *moves* it into the folder instruments/. If instruments/ didn't exists, drums/ would have been *renamed* to instruments/

---

[29]`sites.ualberta.ca/dept/chemeng/AIX-43/share/man/info/C/a_doc_lib/cmds/aixcmds3/mv.htm`

You already have _multiple_ files and you want to _move_ them to another folder

```
mv <existing-file-name1> <existing-file-name2> ...
        <destination-folder-name>
```

```
mv dogs.txt cats.txt animals/mammals/
```
Takes prexisting files `dogs.txt` and `cats.txt` and _moves_ them to the `animals/mammals/` folder.

You already have a file and you want to _copy_ it to another folder

```
cp <existing-file-name> <destination-folder-name>
```

```
cp dogs.txt animals/mammals/
```
(Notice the different command, `cp`) Takes prexisting file `dogs.txt` and _copies_ it to the `animals/mammals/` folder. Keeps the original `dogs.txt`.

You can also use pattern-matching commands (wildcards) such as the asterisk (∗) and period (.) but these are beyond the scope of this book.

Using `which`.   You may never need to use `which` but it might be useful in the future. `which` is used for finding the location of certain execetuables on your computer. The syntax follows `which <program-to-find>` and outputs the file path of that program.

To test this out, try `which git`, `which bash`, or `which fish`.

Part II

Working Solo

Chapter 3

GitHub Solo

3.1 Navigating GitHub

So what is GitHub? It is a website that allows us to share our files with the world and it widely used by the coding/programming community. With GitHub, any public code can be viewed by anyone. Each person is able to copy the code and mess with it on their own computer, all without the original being changed. But if someone wants to change the original, they can submit a request asking the owners to incorporate their changes. GitHub keeps track of all of this information and can also help with version control. If you make a big mistake, you can revert back to old versions and try again. GitHub works closely with Git (hence its name) and is a nice interface for many common Git functions.

Before attempting to tackle Git, I think it's best to show you the ropes of GitHub. After we get set up here and get more comfortable with the jargon, we can move to Git. In this section we will create a GitHub account and get familiar with the layout of GitHub. Much of what is said in this section and chapter is directly applicable to Git (Chapter 4, p. 45).

3.1.1 Creating A GitHub Account

Creating a GitHub account is easy and free. Just be careful along the way; there are a few things to take into consideration. If you already have a GitHub account, skip to Subsection 3.1.2, p. 28. If you don't, keep reading.

First, navigate to GitHub[30]. Any browser should work. Click on the "Sign Up" in the top right corner. A welome box should pop up. Follow the instructions to create an account:

1. Enter your email

2. Create a password

3. Choose a username. Attention! Choose your username carefully. Jennifer Bryan[31] has some great tips for choosing a username. A few are reproduced here:

   - Incorporate your actual name! People like to know who they're dealing with. Also makes your username easier for people to guess or remember.

   - Reuse your username from other contexts, e.g., Twitter or Slack. But, of course, someone with no GitHub activity will probably be squatting on that.

---

[30]github.com
[31]happygitwithr.com/github-acct.html#username-advice

- Pick a username you will be comfortable revealing to your future [or current] boss.

- Shorter is better than longer.

- Be as unique as possible in as few characters as possible. In some settings GitHub auto-completes or suggests usernames.

- Make it timeless. Don't highlight your current university, employer, or place of residence, e.g. JennyFromTheBlock.

- Avoid the use of upper vs. lower case to separate words. We highly recommend all lowercase. GitHub treats usernames in a case insensitive way, but using all lowercase is kinder to people doing downstream regular expression work with usernames, in various languages. A better strategy for word separation is to use a hyphen - or underscore _. [Again the ideas from File Names With Spaces, p. 5 come back.]

- I would like to add one note to her list: avoid gaming usernames. GitHub is a serious professional community and something like "Xx_sp1cyMU3TARD_xX" would not be appropriate.

While it is possible to change your username later, it is better to not fall back on that as that can cause complications. As an example, my username is "ian-curtis". I would have preferred to have it without any spaces but that was taken. It is short, simple, timeless, and I do not feel bad sharing it with anyone. If you have a very common name, consider using hyphens and underscores or rearranging your name. I could have tried "curtisi" or "curtian" (which sounds like I'm an alien).

Once you finish setting up the account (email verification may be required), navigate back to the home page (github.com[32]) and carry on.

3.1.2 The Profile Page

You should be on the home page of GitHub which might look something like my homepage in Figure 3.1.2.1, p. 28. You may not see much if you just created an account. Probably lots of buttons telling you to get started or take a tutorial and whatnot. Feel free to click on those if you'd like, but I'd suggest following along here first and revisiting those later if you need more explanation.
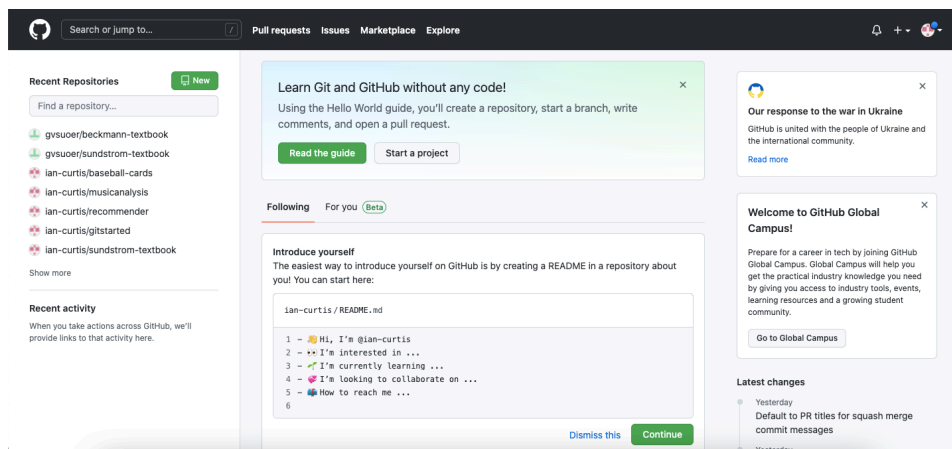

Figure 3.1.2.1 My GitHub Homepage

---

[32]`github.com`

Your profile page can be found by clicking on the robot-y icon (see Figure 3.1.2.2, p. 29) in the top right and clicking on "Your profile". (Notice that the URL uses your username.) You probably don't see much. That's ok, you haven't created anything yet! On this page (the "Overview" tab) you can see your popular repositories, a graph of your contributions and a summary of your activity. Check out my profile page[33] for an example.
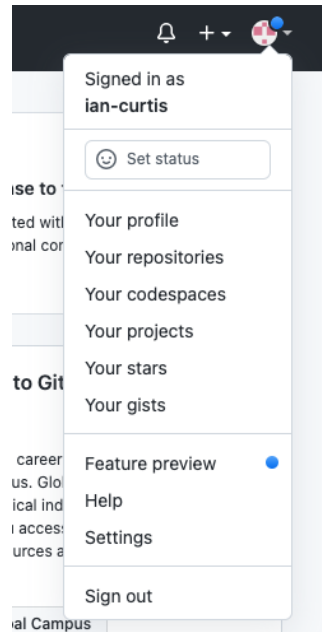

Figure 3.1.2.2 The GitHub Navigation Bar

First, you can edit your basic profile information on the left hand side including your profile image, name, bio, and location. I would recommend filling as much of that out as possible; it allows companies and other people to find you and confirm that you are who you are.

Wait a minute, I just used an unfamiliar word: repository. Essentially, a **repository** is the Git/GitHub way to say "directory" or "folder". A repository is usually one project; one repository contains all of the files and information for one project. For example, the files for this book can all be found under my repository titled gitstarted[34]. This book constitutes as one project and therefore all files for this book are there. Remember Jennifer Bryan from above? All of the files for her book, *Happy Git With R* can be found on her repository[35]. Soon, you too will have repositories! (Note: a repository is often shortened to just **repo**.)

There are a few other tabs at the top of the page. The "Repositories" tab simply lists all of your repos. (You can also find this page from the drop-down menu by your profile image in the top right.) The other three are not important for basic use (I haven't even used them yet).

3.1.3 Settings and Customizations

Click on the profile icon in the top right (Figure 3.1.2.2, p. 29) and select "Settings". Don't be scared of the long list of possible customizations that are available. Most are only applicable to experienced GitHub users. I will introduce the settings you may be interested in right now. The headings of the following paragraphs match up with the headings of the tabs on the GitHub page. Note that some tabs have been excluded as well as some settings within

---

[33]github.com/ian-curtis
[34]github.com/ian-curtis/gitstarted
[35]github.com/jennybc/happy-git-with-r

some tabs. This does not mean that they are not important; it just means that I don't think they are necessary for a beginner.

Profile.   Here, you can change your name, public email (if you have multiple AND if you have chosen to make your email public), a mini bio, relevant info about your other social media and company, location, and whether or not you are searching for a job. See Emails, p. 30 for more info.

Account.   Here is where you can change your username. *I would strongly recommend against changing your username except in dire circumstances.* This could lead to broken links or, if someone claims your old username, links to the wrong repository. This is why I hope you took the time to really think of a good username when created your account.

Ok fine. If you have just created your account, have no repositories, have done pretty much nothing on GitHub other than editng settings, AND are truly unhappy with your username, change it now. NOW. Then try not to do it again.

This tab will also let you delete your entire account. It is in red for a reason. I would highly recommend never deleting your account. Even if you are done with GitHub and don't plan on ever using it again, others can still benefit from your code. Obviously some exceptions could apply here but for most purposes, you probably shouldn't delete your account.

Appearance.   Here you can change how GitHub looks. Is white annoying? Switch to a dark theme. There are three of them. You can choose to have it change automatically with your computer (assuming your computer changes automatically). You can also edit the default emoji appearance.

There is currently a feature in testing allowing a theme designed for colorblind users. To enable this, click on your avatar in the top right and select "Feature preview", then "Colorblind Themes", then "Enable". If you have feedback to give GitHub, you can also do that.

Account security.   Change your password here and you can enable two-factor authentication. You can also see where you are currently logged in. If an unfamiliar session is active, I would recommend changing your password and/or enabling two-factor authentication.

Billing & plans.   GitHub offers different paid plans[36]. Each of the plans has its own benefits (e.g., more storage, more collaborators) but the GitHub essentials work perfectly fine under the free plan and many people do not upgrade. Once you become more familiar with Git, GitHub, and start developing more projects, you might like an upgrade. However, there is no need to pay anything at this point.

Bonus! Students get free GitHub Pro and access to a multitude of extra deals with their Student Developer Pack[37]. Teachers[38] and Schools[39] can also find benefits (some require a payment).

Emails.   You can choose to have multiple emails on your GitHub account. This may be something you want to do as you can also select to have one of your emails be a backup email for security. You can also select if you would like to keep your emails private or not.

---

[36]github.com/pricing
[37]education.github.com/pack
[38]education.github.com/teachers
[39]education.github.com/schools

Notifications.   Pretty simple here. You can choose which notifications you would like to receive. Maybe keep the defaults for now and you can edit them later when you get an idea of what exactly GitHub sends notifications for. I still have the defualts selected and I don't get many emails at all.

Repositories.   If you just created your account, this is probably empty. That's ok! It will be populated soon with a list of all your repositories and repositories you have access to. I do want to point out the first setting, "Repository default branch". It may say "master" in the text box. If so, please change it to "main" (by typing and clicking "Update" which is the new Git/GitHub standard.

For more information on the transition from "master" to "main", please visit TheServer-Side[40].

Blocked Users.   Hopefully you won't ever have to, but if you need to block a user, you can do so here.

Hopefully you now have an idea of how GitHub is organized and are aware of the customizations you can make. However, we have barely scratched the surface of what GitHub can do. In the next section, you will create your own repository and learn some GitHub vocabulary.

3.2 The First Repository and File

Buckle your seatbelts! Here we go into GitHub and the first repository. In this section, we will create a repository with some information about ourselves. I recommend that you don't just read the book; follow along with your own account. Practice makes improvement!

3.2.1 Creating a Repository

There are a few different ways to create a new repo (remember this abbreviation? see Subsection 3.1.2, p. 28).

1. In the top right of the page, next to the profile avatar we saw in Section 3.1, p. 27, there is a plus (+) sign. Click on that and then "new repository".

   Note: GitHub differentiates bewteen a repository and a project. Don't get confused. We want repository (and you will want that for most purposes).

2. From the GitHub homepage[41] there is a section on the left called "Repositories". Next to the title is a green button that says "New". Click on that.

3. If you navigate to the profile avatar in the top right, and click on "Your repositories", there will also be a green "New" button near the top.

Each of these three methods will get you to the same page. It doesn't matter which one you click.

---

[40]theserverside.com/feature/Why-GitHub-renamed-its-master-branch-to-main
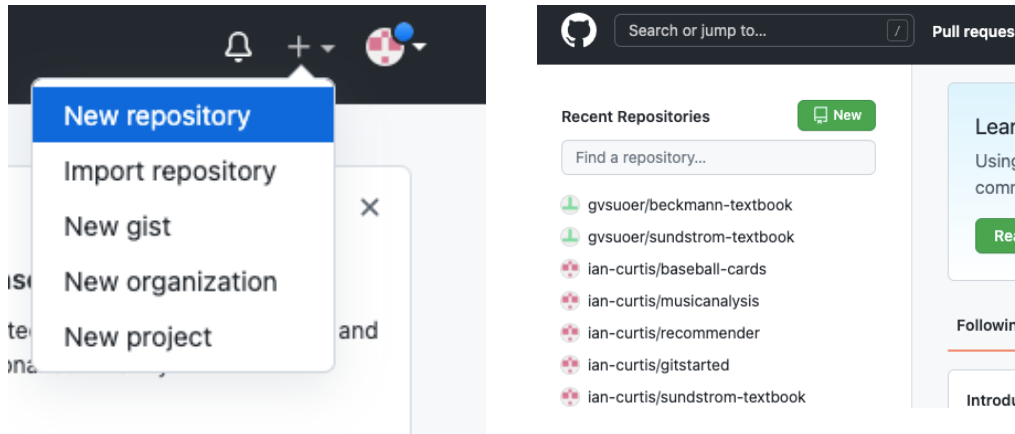[41]github.com

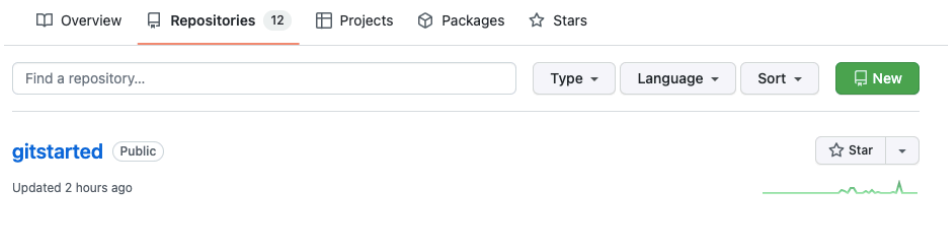**Figure 3.2.1.1** Various methods to create a new repo

Figure 3.2.1.2 Another method to create a new repo

Checkpoint 3.2.1.3 Let's begin the process of creating our first repo. Use one of the methods above get into the "Create a new repository" screen (Figure 3.2.1.4, p. 32).
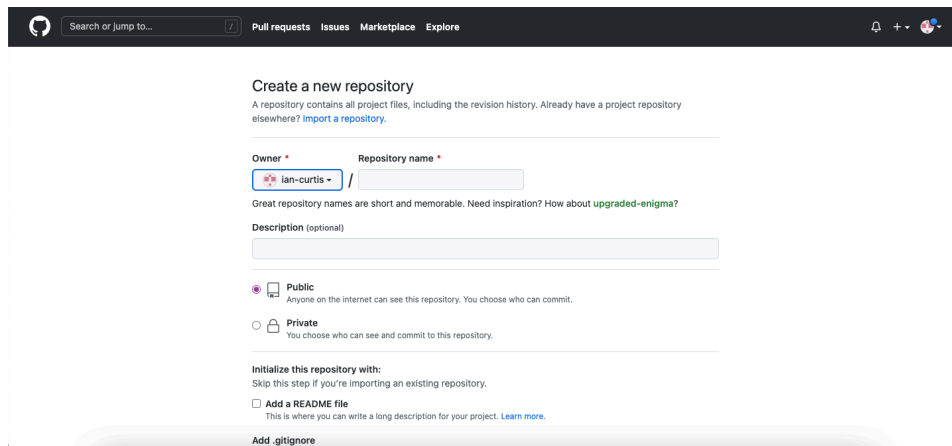

Figure 3.2.1.4 The New Repository Screen

It is not difficult to create a repo. It is difficult, however, to come up with a repo name. Repository name requirements are similar to the conventions for file and folder names (Section 1.1, p. 3) in that they can have no spaces and should briefly describe what the repo is for. *Please take the time to think about your repo names before you create them.* Renaming repositories is possible, but can cause complications in the future (such as broken links). Again, names should be short, to the point, and descriptive. The repo name for this book is "gitstarted", the shortened title.

If you are thinking "Wait! I don't know the details of my project!" then you are not

alone. The problem with this is that you have to decide what your repo is going to be about before you name it. I have definitely been victim to this and have made some rather stupid names that I should've made better. For right now, I will tell you what to name your repo. In the future, it might be best to start with a private repo and then change the name (if you need to) before you make it public. Otherwise, just be aware of the consequences should you change it at any other time.

In the "Repository name" box, type the name `aboutme`. Adding a description is optional and is not necessary for us right now. Next is the viewability. You may choose to make your repos private (only you and authorized collaborators can view/edit) or public (everyone can view and attempt to edit, only you and authorized collabortors can approve edits). It does not matter what you select for this; do whatever your heart desires.

The next step deals with certain files that can be automatically added to your repo. You have the choice for either or none of a `README` file, a `.gitignore`, and a license. A **README file** gives all the pertinent information about your repository in an easy-to-read format. It is often the first thing that people visiting your repo will read and is very important. Check this box and we will come back to the `README` file in just a minute. A **gitignore** file tells Git all of the files *not* to put online. Do not check this box for now; it will be very useful later on, but is not needed now. You can also choose to put a **license** on your code which lets users know what they can and cannot do. If you would like to explore the different licenses, please visit the GitHub documentation[42]. Since we will delete this repo later, you do not have to select this option.

You now should have entered "aboutme" into the name box, selected if you wanted the repo to be public or private, and checked the `README` box. Now, click the green "Create repository" button to finalize the process.

3.2.2 Exploring the Code Tab

Whew, we made it! A new repo has just been created...aaaaand a lot of new buttons and options just appeared. A lot of new unfamiliar words and tabs. Please don't give up here! I will go through all of the important aspects. Many of the pieces are not applicable to beginners and some will not be applicable until we start working with Git and GitHub together. In this section, I will only be explaining the key features and will slowly add in more as we learn more about GitHub.
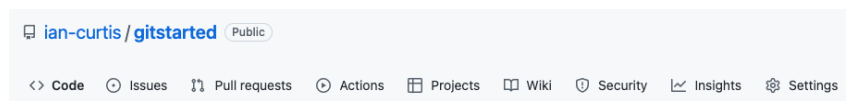

Figure 3.2.2.1 The main new repository contents

Assuming you didn't click on anything, you should see a pretty empty page. In the top left, you should see your username and your repo name. This is common throughout GitHub so you always know where you are. Underneath that is a set of tabs that can be used to navigate through the repository-level settings and options (see Figure 3.2.2.1, p. 33). You should currently be on the "Code" tab. As may be expected, this tab displays all of the files associated with the repo; it is the "home page" for every repository. You should see that a `README.md` file shows up. Also notice that the contents of the README are displayed under the list of files. GitHub knows how important the READMEs are to explaining code that it automatically displays the contents of the file. Right now, ours is empty (except for the default title).

To summarize, this is all you should care about for now:

---

[42]docs.github.com/en/repositories/managing-your-repositorys-settings-and-features/customizing-your-repository/licensing-a-repository

- The navigation at the top

- The content of the "Code" tab

- Noticing our README file in the list of files

- Noticing the contents of the README file underneath the list of files

3.2.3 Editing a File

Well, we have a README file, let's edit it! GitHub makes it rather simple to edit files. There are a two main ways to edit the README file on GitHub:

1. Click on the file name `README.md` in the list of files. Click on the pencil icon on the right of the header of the README file.

2. Click on the pencil icon found along the header of the README file (under the list of files).
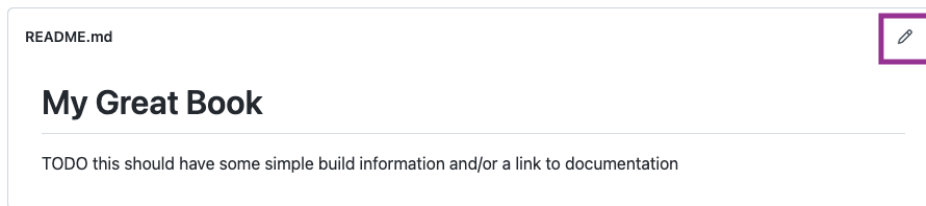
Figure 3.2.3.1 One way to edit a README file
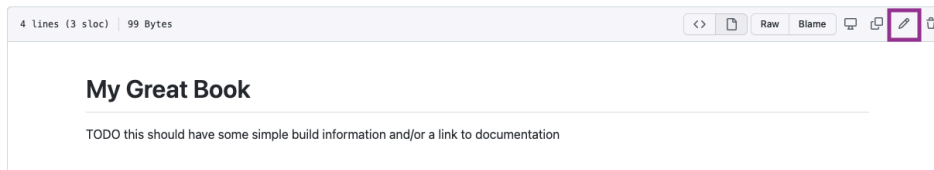
Figure 3.2.3.2 Another way to edit a README file

Checkpoint 3.2.3.3 Use one of the methods above to open the README file and get to its editing screen.

As discussed in Section 1.2, p. 9, a file ending with `.md` is a markdown file. All README files should be markdown files and are able to understand markdown syntax. Essentially, markdown allows us to communicate basic word processing functions simply by typing certain symbols. Have you ever noticed that you can't bold or italicize text in Google? Control (Command) + B or I don't work there! They also won't work in markdown files, unfortunately. But that doesn't mean we can't bold or italicize text. I will not discuss everything on markdown here, so some basic functions are below. It will be in your best interest to become familiar with markdown and its more advanced features sooner rather than later.

Markdown Basics.

The most basic markdown tool is simple text. Just type like you normally would into the text field and the result will be normal, unformatted text. However...

- To italicize a word or phrase, put one asterisk around it: `*I'm italicized*`

- To bold a word or phrase, put two asterisks around it: `**I'm bold**`

- To create headings, place hashtags (pound signs, if you insist) and a space before the text you wish to be in the heading: `# A Title`. The more hashtags you add, the deeper the heading level. Thus, `## A Subheading` is a level two heading whereas `### Another Subheading` is a level three heading.

- To create a bulleted list, start a line with an asterisk (∗) and a space. For numbered lists, start a line with a number and a period (and a space).

- To put words into a code cell, place a backtick around the code: `` `import requests` ``. (The backtick is located to the left of the 1 key on American keyboards.)

You can also add images, gifs, links, links with text intead of urls, tables, and multi-line code chunks. For more information, see Basic Markdown Syntax[43] and Extended Markdown Syntax[44].

It is also important to note that GitHub Markdown differs slightly from other flavors of Markdown. See a nice table from GitHub user vimtaai[45].

Checkpoint 3.2.3.4 Let's practice your markdown skills! Here, you will make the "aboutme" repo name true. Feel free to use the guidelines and sources above. You could also search online if you aren't sure how to do what you want to do. I have provided my (basic) example in the repo for this book[46] (titled `README_aboutme.md` in the `book-activities` folder). But be creative! There aren't any "right" answers here.

Github is pretty cool in that it will show you realtime updates of what your Markdown file will look like after you're done typing. There is a "Preview" tab right above the first line of your README file (you are currently on "Edit file"). After each task below, check the preview to verify that what you typed worked as expected.

(a) Currently, the title of the README is the repo name. Change the title to a more human-readable one.

(b) In the first paragraph, type your name (or your username), how many pets you have, and your favorite hobby.

(c) In a second paragraph, type a sentence stating who your favorite music artist is. Then, make an ordered list of your favorite songs from that artist.

(d) Create a heading level two titled "Oh Look, More Things About Me"

(e) In a third paragraph, write two sentences with two cool facts about you. These could be made up, no one will know! After those facts, create an unordered list of your favorite musical artists after the one in the second paragraph.

(f) Go back to the previous paragraphs and bold two words and italicize two words.

(g) Extra Credit: Add an image, a nested list of your favorite colors and why they are your favorite, a link to your favorite YouTube video, and, if you know a coding language, some code.

Don't leave your file yet! There's one more thing left to do.

---

[43]www.markdownguide.org/basic-syntax/
[44]www.markdownguide.org/extended-syntax/
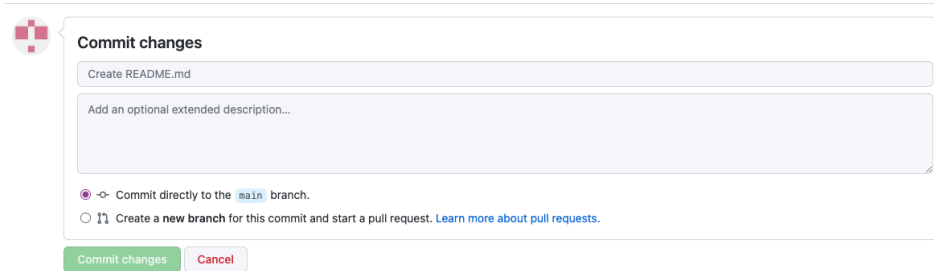[45]gist.github.com/vimtaai/99f8c89e7d3d02a362117284684baa0f

Figure 3.2.3.5 The GitHub Commit Changes Dialog

Right now, we have only edited the file, but we haven't told Github that we wish to keep our edits. It's as if we edited a document, but never clicked save! But if you scroll down to the bottom of the page, you'll see that there is no "save" button. You should see a green "Commit changes" button, as in Figure 3.2.3.5, p. 36. This is important. Github uses **commit** when indicating that you are about to officially upload your changes to your repo. This is your last change to change anything. When you think you are done, go ahead and commit your changes (commit directly to the main branch). Your changes should be represented. Navigate back to the Code tab to prepare for the next section.

Take care to notice the difference between committing and saving. Saving is easy. You can add a word, then save. Add a word, save. And so on. Think of commiting as adding a word, saving, *and closing the document.* Now everytime you want to add another word you have to open the entire document again, add the word, save, and close the document again. This system is put in place to help with version control. Instead of there being one version for each new word you add, there is one version for each commit. Each commit represents a stamp on the version control timeline. When coding, you save your files, but commit your changes. This will become more apparent when we work with Git and with multiple files.

It might be worth saying again just to make sure you've got it. Instead of there being one version for each new word you add, there is one version for each commit. Projects are typically broken up into tiny pieces. Each of these pieces would correspond to a commit. So if something goes wrong, you can revert back to a previous commit. We don't want commits to be for every word since it would take forever to find the commit you want to revert to. Conversely, we don't want commits to contain too many edits as that would require more work if we had to revert backwards.

An Extra Commit Example.

Let's consider one example to help understand the benefits of thinking in commits. Suppose I was working on a small video game and I was tasked with creating different worlds a character could visit. With commits in mind, I decide to break down the task into chunks: I will build one world at a time. When I am done with one world, I will commit. When I am done with another world, I will commit a second time. And so on.

This is different than creating a window in a building in the first world, then saving. Then creating two more windows and a door, then saving. Then adding some a tree and some flowers, then saving. We save much more often than we commit. But *we save using our computer and commit using Git/Gitub.* I would never save to Github or commit to my computer. Saving is light, commiting is heavy.

That being said, do make sure to save your files! You want to save your progress on your computer, but there is no need to commit after each time you save.

---

[46]github.com/ian-curtis/gitstarted

You know should feel slightly more comfortable with the Code tab and editing a file on Github. The next chapter will look at branches which help with adding new features to our code.

3.3 Working With Branches

Suppose you were writing a three-stanza poem for a contest. The final product must be handwritten on fancy letterhead, in high-quality black pen. As multiple drafts go by, you finalize the first and the third stanza. You're so confident in them that you write them on the letterhead in pen. But you're stuck on the second stanza.

After some introspection and brainstorming, you finally come up with two potential second stanzas. But you can't decide between the two! You decide that the best way to figure out which one you like is to write each one in the poem and see if it flows with the rest. You can't just write one in the official poem (in pen!). What if the one you choose isn't what you like? Then you have to rewrite the entire poem.

Instead, you decide to get out two pieces of scratch paper and write the poem in pencil and with a respective second stanza. That way you can hold them side-by-side and compare their flow. This method makes it easier for you to decide which of the two candidates you prefer. Once you have done so, you write the final stanza in pen on the letterhead and recycle the scratch paper with your drafts.

Believe it or not, this potentially unrealistic example is a great demonstration of what Git and Github call **branches**. Branches will be used extensively in the coming sections and chapters and are quite important to smoothing out your workflow.

3.3.1 Thinking About Branches

Github organizes your edits and changes in a tree-like format. The official, published version of your files always lies on the **main branch**. You could think of the main branch as the trunk of a tree. When we created our README file back in Section 3.2, p. 31, we started on the main branch, edited a file, and commited right back to the main branch. It was as if we wrote something new in pen on our official letterhead. See Figure 3.3.1.1, p. 37 as a visualization of this type of commit.
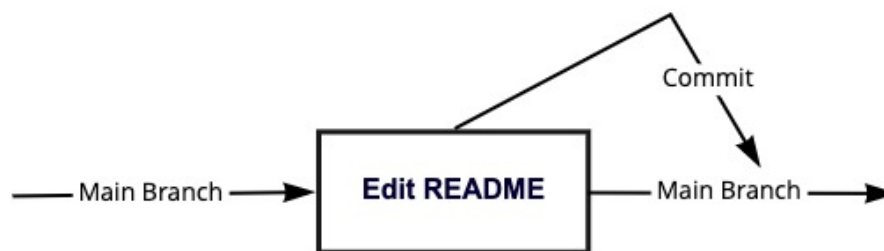

Figure 3.3.1.1 Diagram of a commit from `main` to `main`

This type of editing is usually discouraged. It is best practice to create a new branch, made some edits there, then submit a pull request to merge your changes into the main branch.

Whoa. I said a lot of stuff there. Lots of new words and things you don't know how to do. Don't worry, I'll guide you through it in this section. When you get a new project, you want to try and break it up into pieces. Each piece will have its own branch on the Github

tree. For instance, the two poems that you wrote on scratch paper above were two branches off of the official poem. They each had the same first and third stanzas; only their second stanzas differed. See Figure 3.3.1.2, p. 38 for a visualization of branches in this instance.
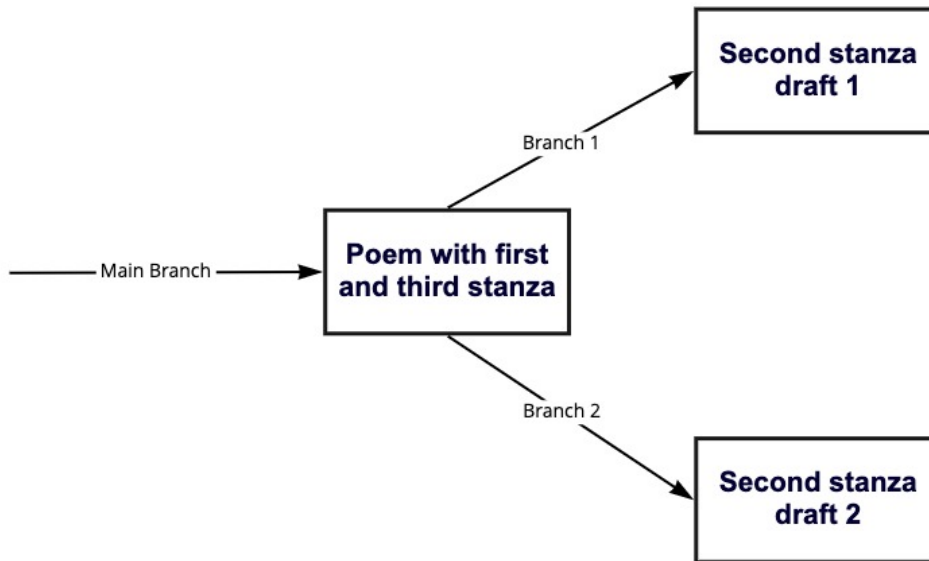

Figure 3.3.1.2 Diagram of two branches off of main

Notice how in Figure 3.3.1.2, p. 38 that the main branch does not move forward when branches are created, it stops once a branch is made. When you first create a new branch, all you have is a copy of the main branch. However, when you edit your new branch, the changes do not appear into the main branch. Branches are like testing grounds. Try some new things, see if you like them, see if they work, then make it official. "Making it official" with Git and Github is called **committing** (as we have seen). Two new terms that go with commits[47] are **merge** and **pull request**. Oftentimes, you will hear people say that they just merged their changes into main. This means they committed their changes to the main branch and updated the official version. With the poem example, there are two possible merges you could make into main, as diagrammed in Figure 3.3.1.3, p. 38.
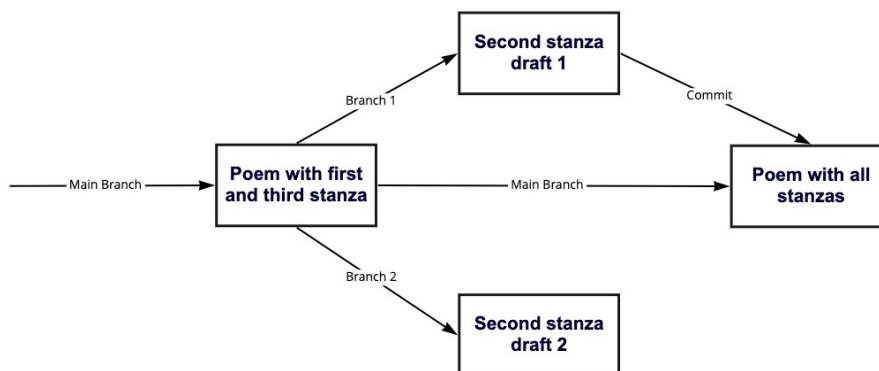

Figure 3.3.1.3 Diagram showing a possible merge into main

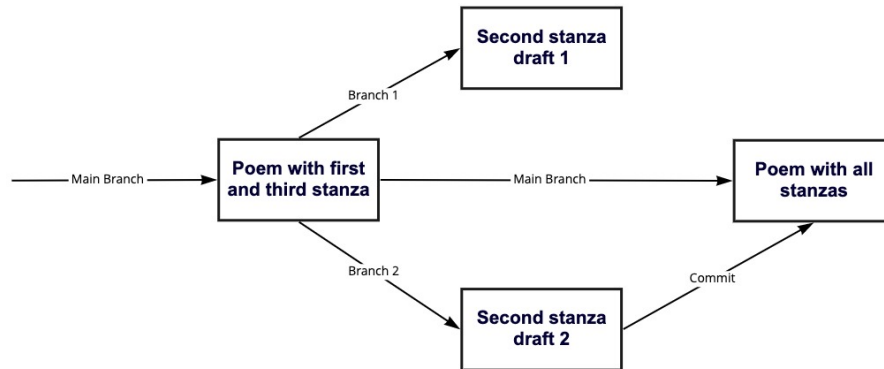---

[47]Notice how "commit" is both a noun and a verb.

Figure 3.3.1.4 Diagram showing another possible merge into main

You will also often hear about pull requests. Recall how GitHub allows anyone to edit anyone else's code, but in order to change the official version, they have to ask for permission. The asking for permission is sending a pull request. You are asking the creator of the main branch their permission to pull in your changes. Don't worry too much about this now; we will cover it when we get to multi-person collaboration. You will see the term come up in this chapter however which is why I chose to introduce it here.

3.3.2 Creating a Branch

Creating a branch is not difficult. To do so, first make sure you are on the "Code" tab of your repository titled `aboutme`. Below the list of tabs, you should have a row of items. The first is a button with a branch-like icon with the word "main". Next to that, there is the same branch-like icon with "1 branch" and then a tag-like icon with "0 tags" (Figure 3.3.2.1, p. 39). [48] Now,

1. Click on the word "main". This will open a drop-down menu with all of your current branches (Figure 3.3.2.1, p. 39). You probably only have one branch: the main branch.

2. We will be adding a part to our README file: our favorite animal. Type `animal` in to the textbox that appears. Your cursor should navigate their automatically.

3. Since we don't have a branch named `animal`, GitHub asks us if we want to create a new branch with that name. That is indeed what we want to do. Click on "Create branch: animal from 'main'"
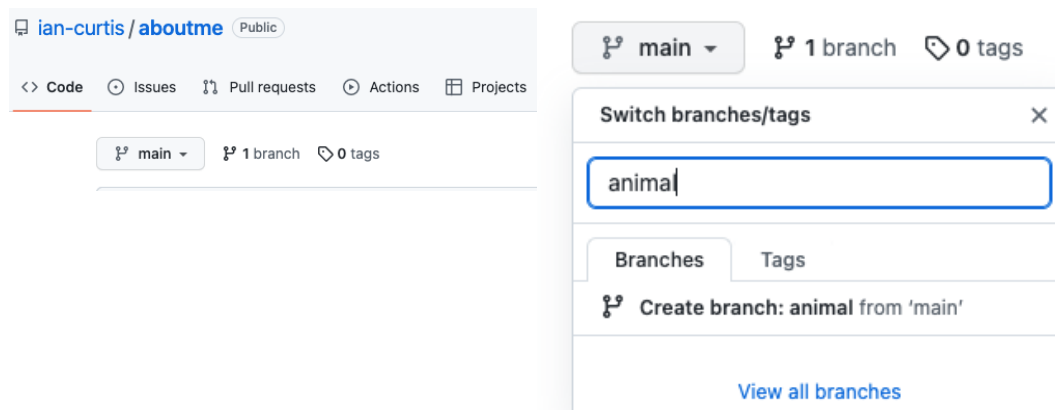

Figure 3.3.2.1 Reference for creating a new branch on GitHub

[48]Your numbers may be different if you played around with GitHub before this step.

Once you have created the new branch `animal`, your screen will have changed slightly. The branch-like icon with the word "main" has changed to say "animal". This is how you will know what branch you are looking at. Also notice how it says we have "2 branches". We do: main and animal. Right underneath those items, you may also notice GitHub has placed a box stating that "This branch is up to date with main.". This is what we should expect. This statement indicates that the `main` branch and the `animal` branch are identical (see Figure 3.3.2.2, p. 40).
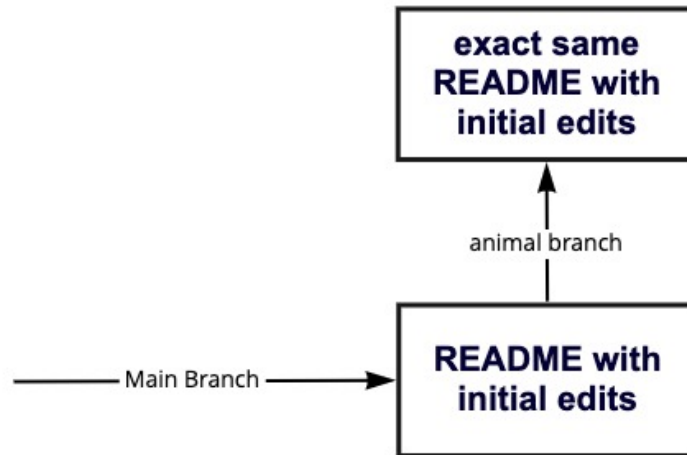
Figure 3.3.2.2 Diagram mapping branches after creating the animal branch

---

A Few Notes on Branches.

- Take caution when naming branches. You should follow the same guidelines the were discussed with file and folder names in Section 1.1, p. 3.

- Branch names should be extremely descriptive in the shortest way possible. It is wise to break up your project into distinct chunks and assign one branch for each chunk.

- It is very easy to get lost within branches. You can create sub-branches off of normal branches. You can delete branches or ignore them. I advise only creating the branches you need at the current moment. If sub-branches (i.e., sub-tasks) are something you are interested in working with, I would recommend keeping a picture of the relationship between branches.

- Once you are done with a branch, delete it. Otherwise, it will clog up your branch list. You can always create it again.

3.3.3 Working On a Branch

We named the new branch `animal` to describe the changes we plan to make: add our favorite animal. Editing a file on a branch is no different than editing a file on `main`. However, we now have a safeguard. We can edit and commit as much as we want to the `animal` branch and never make any changes to `main`. Returning to the analogy at the beginning of the section, we no longer have to write in pen on fancy paper unless we are sure we want to change something.

Hopefully you are able to see the power in this. In a more serious project, commiting

too early could have much bigger ramifications. Suppose I am coding a new character for a video game and the character has to use vulgar language. In a simple workflow, I would create a branch called `vulgar` or `new_npc` with a sub-branch of `language`. Then, I would incorporate my code on my branches. Once that was done, I would run my code to test that it worked. Then I would submit a pull request, requesting my boss to pull in my new changes into the official version of the game. They would likely take a look at my code and try it themselves before accepting the changes. But let's say I forgot to add in the censorship beep over the language. If I had just merged right into main, my mistake may have been missed and I could get in serious trouble. But since I worked on a branch, my forgetfulness can be caught before the changes get incorporated and there won't be any flustering to quickly revert changes or fix the error.

Checkpoint 3.3.3.1 Editing a File On a Branch. Exploring the ideas in Editing a File, p. 34, add a level two heading to your README called "My Favorite Animal". Underneath that, type your favorite animal. Do not commit yet.

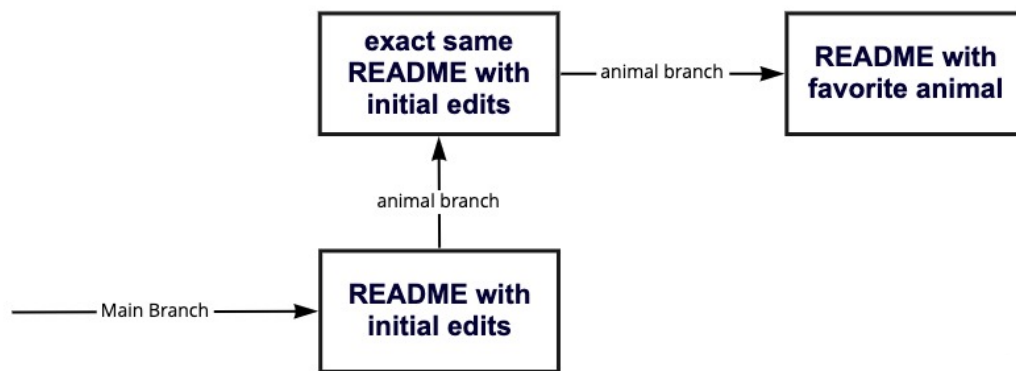The current state of the branch is shown in Figure 3.3.3.2, p. 41.

Figure 3.3.3.2 Diagram showing branches after add a favorite animal

After you have made some changes, scroll down to the "Commit changes" section. Now is the time to introduce commit messages. A **commit message** is a short, but informative (notice a recurring pattern?) note of what your changes do. You may be tempted to skip this when working with yourself. Please try to write the best commit messages *at all times*. When you look back at your repo in the future, it will be helpful for you to remember what you did. Commit messages also help others who may look at your project know what you did in each commit. You may see my various branch names and commit messages at the commit list for this book on GitHub[49].

Let's create a commit message for our changes on the `animal` branch. In the "Commit changes" block, there is a text box that is autofilled with "Update README.md". This is the default and for some instances may be enough. But I have the tendency to write slightly more detailed messages for my commit and I recommend you do so as well. Instead, type "add favorite animal". There is no need to add the file name as this message will only be tied to the file(s) you have edited. (Later on, we will see that specific commit messages are only associated with the files you have staged and committed but that's a topic for Chapter 4, p. 45.)

---

[49]github.com/ian-curtis/gitstarted/commits/main

A Note On Commit Messages.

It is up to you how to format your message. Some people write in complete sentences while others use incomplete sentences with capital letters. Some people use past tense, others present tense, and even others use present participles (add*ing*). I choose to use present tense with incomplete sentences and lowercase letters. But that's my preference. You should find what you like and stick with it.

...Except when you're editing someone else's files. In this case, pay attention to how someone else words their commit messages and imitate them. This will make life easier for the owner of the files and for anyone else who looks at the files in the future. You may even find that your commit message is changed by the owner of a repository to match their guidelines.

It is not necessary to fill out the description text box (feel free to do so if you would like!). Ensure the "Commit directly to the `animal` branch." is selected, then click "Commit changes".Your file should be updated! Just remember, *you have only updated the copy of the file on the `animal` branch, not `main`*.

Navigate back to the code tab to continue. Make sure you are still on the `animal` branch. You can verify this by looking at the drop-down menu text or by checking the contents of your README file (if your favorite animal is there, you are on the `animal` branch).
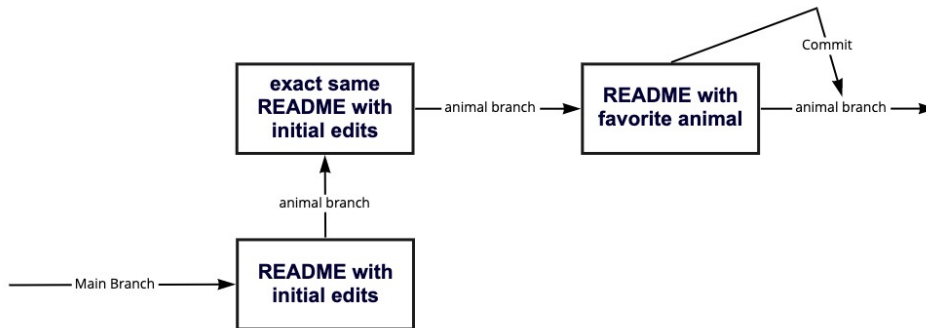

Figure 3.3.3.3 Diagram showing our commit from `animal` to `animal`

3.3.4 Creating a Pull Request (with Yourself)

Alright, so you've edited a file on a branch AND you are happy with your changes. Everything looks good and works well on the branch so you are ready to incorporate your changes into the official version of your project. This is called creating a **pull request**. It may sound odd to call it a pull request when it seems like you are actually *pushing* your changes toward the main branch. However, often with pull requests when collaborating, you don't have permission to access the main branch of another repo, so you have to request that someone *pull* your changes into the main branch.

I admit, it is silly to have to work with pull requests when working solo. Why would I need to ask myself permission to pull in my own changes? But pull requests are essential when working with multiple collaborators and I recommend using branches and pull requests in your own individual work for practice and good habits.

Pull requests are only necessary when using branches (recall how when we edited the README file directly from the main branch we had no pull requests). We are currently in a situation that would require a pull request. How do we create one? First notice (on your GitHub repository for the `animal` branch) how a new message has appeared. This probably

says

> This branch is 1 commit ahead of main.

This is GitHub's way of telling you that the branch you are currenlty on has changes that are not reflected on the main branch. Specifically, we have committed one time and the changes from that single commit do not appear on main. We could go and edit the README file some more and commit those changes. In that case, we would be 2 commits ahead of main.

The dialog box that contains this message also has another feature: a "Contribute" option. If you click on this option, a box pops up telling us again that we are 1 commit ahead of main. In green is a button inviting us to "Open pull request". Click on this.

There is a lot on this page but most of it will be dealt with later. For right now, notice that GitHub tells us which branch we are trying to send to main at the top. `animal` should be on the right and is pointing towards `main`. Next to this, you should see a check mark and an "Able to merge" message. This is good; see Section C.7, p. 96 if your branches cannot be automatically merged.

Your cursor should have automatically navigated to a dialog box. This has defaulted to our latest commit message. You should make sure that you have put an informative message on what your pull request accomplishes as a whole. Remember, it is possible to have multiple commits before opening a pull request so you could have adjusted a few things. For us, "add favorite animal" is sufficient and we can open the pull request. Click on the green "Create pull request" button.

Nice, another new screen. This time notice how we have changed tabs. We are now in the "Pull requests" area, which should make sense. The pull requests page is for more than just accpeting pull requests. If someone else has contributed to your repo and has opened a pull request, you can start a conversation here. Suppose you have a question about someone's changes or think they should improve their addition before you approve it. You can say that here.

But we have no problems with ourselves. Our pull request has been created and is waiting for permission to be pulled into main. In some instances, you will have to wait for a repo owner to accept your pull request. However, you are the repo owner and can **merge** the changes yourself. You can do this on the current screen. Find the green "Merge pull request" button. GitHub will give you the chance to edit the message if necessary. Since we have no changes to make, click on "Confirm merge".

Hopefully you were given the "Pull request successfully merged and closed" message. From here, click on "Delete branch". You should have no fear about deleting the branch at this point. Your changes will not be lost because you just incorporated them into `main`. You now have two identical copies of the same files and can get rid of the extraneous copy: your old branch. If it makes you feel better, GitHub keeps track of commits and merges. If you ever want to revert back to an old version, you can in Section C.6, p. 95. For now, go back to the "Code" tab and verify that your favorite animal appears in the README in the main branch.
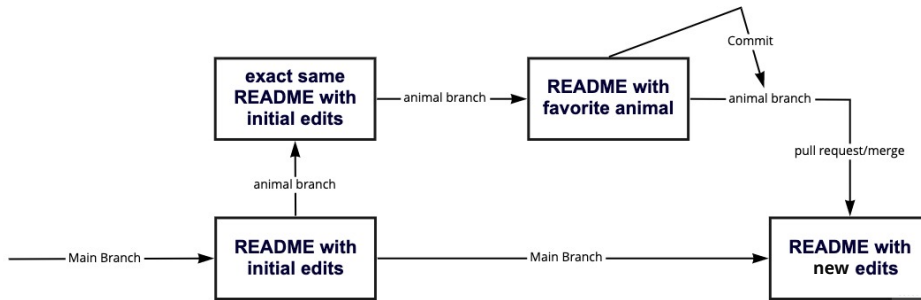
Figure 3.3.4.1 Diagram showing a pull request from `animal` to `main`

Checkpoint 3.3.4.2 Branches and Pull Requests. It is time to put together what you learned in this section and make some edits of your own. Don't be afraid to look back at what we did earlier for reassurance. Start this exercise on the "Code" tab of your `aboutme` repo. The goal of this exercise is to add another section to the README file.

(a) Think about something about you that you have not put in the README yet. Perhaps your favorite food, quote, or historical character. Once you have chosen what you want to add, create a branch with an appropriate name.

(b) Edit your README file. You should have at the very least a new level two heading indicating what you are adding and a piece of text with your addition.

(c) Add an approporiate commit message and commit your changes to your new branch.

(d) Open a pull request to merge your changes into the main branch.

(e) Merge your new pull request into main and delete your old branch.

(f) Verify that your change did in fact get merged into main.

Congratulations! Hopefully you have a better understanding of how branches and pull requests work in GitHub (and their potential).

Chapter 4

Git Solo

Git is difficult to wrap your head around partly because it's new and partly because it's scary. There are many things that could go wrong and that are difficult (but usually not impossible) to revert. But Git's power of version control is so important. The goal of this chapter is to help you get familiar with working with Git on the command line and how it interacts with GitHub.

I assume you have read Chapter 2, p. 15 and Chapter 3, p. 27 before tackling this chapter. The vocabulary used by GitHub is identical to Git's (this is on purpose) and if you understand the basics of how Github operates, you will find some of Git to be simple. Git runs on the command line so it is also important to have some experience with a terminal and how it operates.

I also assume you have followed along with the GitHub activities throughout Chapter 3, p. 27. They are super important for understanding the content and I continue where I left off in this chapter. We will be updating the `aboutme` repository with Git rather than just GitHub.

4.1 Getting Set Up with Git

You should have Git installed at this point. If not, see Materials: Windows, p. xiii, Materials: Mac, p. xv, or Materials: Linux, p. xvii. This means you already have Git ready to go. But how do we use it? We created a repo on GitHub but need a way to connect that repo with our personal computer. That's a great feature of Git: *to connect the files on your computer to GitHub.* It also performs version control and automatically records changes that have been committed.

It's not always feasible to edit file on GitHub, as we did in Section 3.2, p. 31 and Section 3.3, p. 37. Certain text editors (for instance, Visual Studio Code which you might also have installed) have features built in to ease coding and file editing. VS Code extensions might add color to text to help you see what's going on in your code. Others provide bug fixing and error catching before the errors actually occur. In general, you will edit files on your personal computer and will want to share them with the world. *I would stay away from editing on GitHub unless you have a small correction and/or you don't need any special editing tools.*

4.1.1 Cloning a Repository

In order to make Git work, we need to set up a connection between your computer and your GitHub repo. From here on, I might refer to your personal files as your **local files**. A file is typically considered "local" if it does not exist on GitHub (or anywhere else on the internet). It is local because only you have access to it. You will likely hear other people use this term and its derivatives (such as "editing locally"). Whenever it comes up, it refers to work/files/things on your computer and not online.

So let's use the new vocab. We need a way to edit our GitHub repo locally but still have version control and GitHub tools. Git is the right tool for the job. Performing this connection is called cloning. A **clone** of a repository is a local version of a repo that exists on GitHub. Luckily for us, GitHub expects us to do this and makes is rather simple to do.

Somewhat unfortunately, one thing must be done before we are able to clone a repo. Interacting with your GitHub materials in this way will require authentication; in other words, your password. This is necessary for safety and is a smart feature. However, Git/ GitHub has deprecated the use of passwords on the command line so we must generate an Personal Access Token (PAT). The following procedure guides you through this process. I recommend creating a new PAT for each repo to ensure that if anyone gets unauthorized access of one repo, they won't be able to get to your others.

Git Procedure 4.1 Generating a PAT.

 (a) Click on your GitHub profile icon and navigate to Settings.

 (b) Scroll down and click on "Developer Settings".

 (c) Click on "Personal access tokens". This page will give you an overview of all the tokens you have created, their expiration dates, and the things they have access to.

 (d) Click on "Generate new token". Your GitHub password may be required.

 (e) Enter a title for your token. What is your project? This name doesn't really matter; it exists solely to help you remember why you made it. If this is your first PAT, perhaps name it "Git Started Work" or "About Me".

   If you choose the above, make sure you only use the token for that purpose!

 (f) Choose an expiration date. At what point do you plan on being done with this project? Set the date for a month after that. Don't fret too much about it; you can always create a new PAT later for the same project. But I can bet that you won't want to do this process more than once for one project if you can help it.

 (g) Select a scope or scopes. What do you want this PAT to give you access to? This will depend on your project but less is better (for privacy in the event of hacking). Don't be super concerned as it is difficult to hack PATs. If you do not give a PAT permission to do something, that something will not be able to be done locally, even by you.

   For this project, you will want the "repo" scope and maybe the "user" scope. Once you have the scope(s) selected, click "Generate token".

 (h) You will get an email that a PAT was created. Record your PAT somewhere. You will never see it again after you close or refresh the page. Don't put it somewhere easily accessible. *Treat PATs like passwords.* They are confidential and should not be shared. When Git (on the command line) asks for your password, you will enter in your PAT instead.

Now that we have a PAT, let's work on cloning a repo! You will likely need your PAT in this procedure. Note that sometimes Git remembers old PATs. If Git does not ask you for

a password, this doesn't mean something went wrong, it just means that you are already authenticated.

Git Procedure 4.2 Cloning from GitHub. This procedure assumes you have a repo on GitHub called `aboutme` and have been following along with the changes as described in Chapter 3, p. 27.

(a) Navigate to the Code tab your repo `aboutme`. Make sure you are on the main branch.

(b) Find the green dialog box labeled "Code" (yes, the Code box inside the Code tab). Notice you are able to add files manually through upload or through GitHub. You certainly could do this but you will most likely be ignoring the version control and the wonderful experience of using Git!

Click on the green Code box. This box is titled "Clone" which is a good sign. A url-like string should appear (if not, be sure you have the HTTPS tab selected.) Copy this string.

(c) Open your terminal. Use `cd` to navigate to a folder where you want your repo to live locally. Remember that repositories are essentially folders. When you clone your repo, it will look like a folder on your computer. In some cases, it might not make sense to create a folder and then clone your repo as it will look like a folder that contains a folder. Note that the name of the cloned repo/folder will be the name of your repository.

(d) You should be at the location you want your repo to be. It is now time to activate Git. Every time you use a Git command, you start with `git`, then the command. For cloning:

```
git clone <copied-https-string>
Cloning into 'aboutme'...
remote: Enumerating objects: 17, done.
remote: Counting objects: 100% (14/14), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 17 (delta 2), reused 0 (delta 0), pack-reused 3
Receiving objects: 100% (17/17), 5.06 KiB | 1.01 MiB/s, done.
Resolving deltas: 100% (2/2), done.
```

where <copied-https-string> is replaced with the HTTPS string you copied in Task 4.2.b, p. 47. Press enter to run the command.

Your terminal may ask you for your GitHub username and password. This is ok: enter your username and instead of typing your normal GitHub password, paste/type your PAT as found in Git Procedure 4.1, p. 46.

(e) Verify that the clone worked. In the location you chose (on your computer), you should see a new folder with the name of your repository. Inside of that folder will be your files!

Cloning a repository does not navigate you into that repo. In other words, when you clone a repo, you won't see the files in the repo until you `cd` into the newly-created directory. In order to use Git, you will have to navigate your terminal into the repo. In this instance, you would do `cd aboutme`.

4.1.2 Branches With Git

Great! We've cloned a repository. Now what? You probably want to jump into editing files but wait! Best practice says to create a new branch first. Sure, you are working solo (if you

aren't, see Part III, p. 63) and you can really do what ever you want. You could just edit everything on the main branch and never have to worry about other branches. No one will be able to screw with your files without permission.

No one except you! Yes, you could make a mistake and break your program or project. You want to avoid that. I would recommend *always* using branches whether you are working alone or not. If you make a horrible mistake, branches help you fix mistakes without messing with your main branch. Remember, if your repo is public, anyone can access it. Anyone can view the files, download the files, clone your repo, or fork your repo (Subsection 5.2.1, p. 64). You want to make sure that the files on the main branch are up-to-date AND functional at all times. Branches will let you test new things without breaking main.

We saw earlier how to work with branches on GitHub, but you can also do this right from the command line. This can be done with one command and doesn't involve navigating multiple pages as we did on GitHub (Section 3.3, p. 37).

Git Procedure 4.3 Branches With Git. Later on in this chapter, you will be adding to your README file. You first will add the country you were born in.

(a) Ensure that your terminal is navigated into your repo. You may have noticed that your terminal changed! The name of your active branch is displayed by the file path. Mine now has a `(main)`. This is how you remember which branch you are editing on.

(b) Remember that each Git command begins with `git` to let your computer know you will be using Git. To create a new branch, use the following:

`git branch <branch-name>`

where `<branch-name>` is the non-spaced name of your new branch. Try creating a new branch now, with the name of `country`. So,

`git branch country`

(c) Well shoot. Git still says we're on the main branch. We want it to tell us that we are on the `country` branch. That's because we need to tell Git what we want to work on that new branch. Git doesn't move you to that branch in case you want to make more than one branch at one time.

With Git, navigating to a new branch is done with `switch`. In general,

`git switch <branch-name>`
`Switched to branch '<branch-name>'`

If done correctly, Git will tell you that it switched to the new branch. This should be reflected in the branch name indicator. Switch to your new branch `country`:

`git switch country`
`Switched to branch 'country'`

My terminal now says `(country)` instead of `(main)` so I have verification that the process worked.

Branch/Checkout Shortcut. Using a little terminal trick, we can actually create a branch and switch to that branch at the same time. This uses the `-c` switch. For our example in Git Procedure 4.3, p. 48,

`git switch -c <branch-name>`
`Switched to new branch '<branch-name>'`

By adding the `-c` switch to the `switch` command, Git creates a new branch with the title given and checks out that branch. This is quite useful if you are only creating one branch

and want to switch right to it. If you are uncomfortable with this, then feel free to continue using the method in Git Procedure 4.3, p. 48. Each will get you to the same place.

Note About `git switch`. The `git switch` command was introduced to help remove user confusion. Previously, the command was `git checkout <branch-name>`. However `checkout` has two functions: to switch branches and to reset files (if a file has edits that you want to restore, you can use `checkout` to restore the file to how it was before you made the changes).

    `git checkout` still exists and you are welcome to use it to switch branches (you would use `git checkout -b <branch-name>`). However, to avoid confusion, I recommend using `git switch` for changing branches and `git restore` for restoring files.

You are now ready to edit files! You determined a chunk of your project to start with (adding your birth country), created a corresponding branch (`country`) and checked out that branch. Until you change your branch again, all edits will take place on your current branch *only*. Continue on to edit your README file!

4.2 Editing Files

Admittedly, the title of this section is a little misleading. You can't edit files with Git. Git just keeps track of your files and your changes. Take this section as a chance to familiarize yourself with your text editor (VS Code is recommended and is demonstrated here).

    To recap, we used the command line and Git to start the version control process for our next edits. We activated a branch called `country` which we chose intentionally because we plan on adding our country to the README file. Git will keep the `country` branch "active" until we tell it to switch to another branch. Any changes you make, files you delete, or files you create will be recorded under this `country` branch. In other words, only the `country` branch will change, not `main`.

    But let's try editing our file. This is the main goal after all. Open your README file in your preferred text editor. If you're feeling adventurous, use the terminal to do it. If you are using VS Code, read Opening Files With VS Code, p. 49 before using the terminal to open the file.

Opening Files With VS Code.

Try to open the README file with the terminal. (Remember how? See Section 2.3, p. 17 if not.) What happened?

    Any number of things could have happened. It might have worked and VS Code opened with your file ready to edit. Maybe it didn't work at all and no applications opened. Maybe your terminal opened up a different app. The problem with the `open` command is that it picks the default application and opens that. For me, by default, all `.md` files open in RStudio. So, I have to specifically tell my terminal that I want VS Code to open the file, not anything else.

    Luckily, VS Code has its own command! Remember how every Git command starts with `git`? Well that same idea holds for many different commands. To open a file in VS Code, type `code <file-name-and-extension>` where `code` stands for VS Code. Now things should work correctly and VS Code should open your file.

    Hopefully, the contents of the file look familiar. Everything should be the same from when we edited it back in GitHub. Take a minute and add a heading called "My Birth

Country" and type a sentence stating your birth country. (If you don't want that information on the internet, put any country … maybe your favorite country.) Save your file.

I would like to point out some helpful features that vs Code has (other text editors will likely have these same features but not in the same places). In the bottom left corner, vs Code displays the name of your current active branch. This is incredibly useful in case you forget what branch you are on (especially if you are working with multiple branches) or if you want to verify that you are working on a branch and not on main. vs Code also displays (on the left side) a list of all of your files and directories that are inside your GitHub repository allowing easy access to any file you might wish to edit.

All files you currently have open appear as tabs at the top of your screen. You might also have noticed that when you saved your file, the name of your README changed from white to a cheddar color and an M appeared. vs Code has Git integrated inside of it (one of the many benefits). As Git keeps track of your changes, vs Code tells you which files you have *m*odified (M, yellow), *d*eleted (D, red), or created (U, *u*ntracked). If you were to undo *all* of your changes and saved the file again, the M would disappear and the color would go back to white.

Hopefully editing the file was fairly straightforward. We now have changes that we have made and we want to get those changes back on to GitHub and on the `main` branch. Don't forget that the changes we made in this section are only local and only on the `country` branch. The next section covers how to stage, commit, and push your changes.

4.3 Sending Changes Back To GitHub: The Three-Step Process

Ok, great! We have some edits, but how do we let other people see them? Remember that we have been editing locally so nothing new has shown up on GitHub. The goal here is to share the work with the world, so we need a way to send the changes back to GitHub. Luckily, by cloning our repo, we set up a connection between our local repo and our GitHub repo.

This is a very important section. Each step must be completed in the proper order to avoid Git errors, headaches, and file loss.

With Git, there are three main steps to sending edits back to GitHub. That may seem like too many but in fact each step serves a different purpose and gives you a little bit of freedom with how you edit on a branch. The three steps are

1. Staging your files for sending

2. Commiting your changes

3. Pushing your changes to GitHub

Step 2 may seem familiar. Yep, the **commit** that happens here is the same as a commit we did earlier!

4.3.1 Wait, Which Files Did I Change Again?

This happenes often in large projects (and even in smaller projects). You have a branch, you've been working all day editing and creating files, making sure things work. But now it's time to stage the files and you don't remember what files you've edited! You aren't even positive what the file names are. True, vs Code color codes your edited files but you might have folders and subfolders throught your repository and don't really feel like searching

through everything to record what files you changed and which of them you want to send to GitHub.

Luckily, Git has the command for you! With one line in the terminal, you can easily see a list of files you changed, deleted, and added. Let's explore that with our small case.

Git Procedure 4.4 Trying `git status`. The command you will need is simple: `git status`. The `git` is necessary to tell the computer that we will be using git and the `status` gives the the current status of all of the files in our repository. We might say that it shows all of the files that Git is **tracking**. Open the terminal, verify that you are in the correct directory and on the correct branch and try the command. Did it spit out what you expected?

A Small Note On VS Code. It is worth noting that if you are using VS Code, you do not need to navigate back and forth from a terminal window and the text editor. VS Code has a built-in interface to use your terminal. To access this, navigate to the "Terminal" heading in the navigation bar at the top of your screen and click on "New Terminal". You may also use the keyboard shortcut [Ctrl] + [Shift] + [`] (On a Mac, you would still use [Ctrl] not [command].)

4.3.2 Step 1: Staging Files

The first step in sending files back to GitHub is to stage your files. Basically, this step is like you saying "I have edited some files and these are the ones I would like to send to GitHub". With our small example, this step is a little silly. We only edited one file, of course it's the one we want to send back.

The magic of staging files, however, lets you choose which files you want to push back. Maybe I'm working on three files at once, but only two are ready to go back to GitHub. Maybe I'm writing a book and I want my editor to be able to see chapters 1-4 but not chapter 5 (which I'm currently working on). At any given time, you can decide which files you want to stage.

Git Procedure 4.5 Trying `git add`. All this is great but how do I stage my files? How do I tell Git which files I want to send back?

(a) The output from `git status` in Git Procedure 4.4, p. 51 actually gives a hint on how to proceed. If you haven't completed Git Procedure 4.4, p. 51, I recommend doing so now.

(b) The staging command is done with `git add` and the command is followed by all the files you want to add. There are three cases with this command:

You want to add specific files	Type in each file name and extension individually with a space between each file.
You want to add an entire folder of files	Type in the name of the folder followed by a [/] (e.g., `images/`).
You want to add all changed / created / deleted files	Type in a single period ([.]) instead of file names.

You can run as many or as little `git add` commands as you wish. For instance, you

could do `git add images/ my_file.txt` or you could do `git add images/` and then do `git add my_file.txt` separately. It's up to you and how comfortable you feel with Git.

Try now to stage your README file using one of the three cases above.

Hint. Case 1 or Case 3 will work in this instance.

(c) Use `git status` to verify that your staging worked and that you didn't add any extra files on accident (see Git Procedure C.1, p. 91 if you did).

4.3.3 Step 2: Committing Files

"Commit" is a great name for this step in the process. At this point, you are indeed ready to commit to what you have changed and to send your selected files to GitHub.

Remember committing from Section 3.3, p. 37? This commit is the same concept as it was before: telling Git/GitHub that your changes are done, you are sure they are done, and that you would like them to be a part of your main, active branch. This is easy on GitHub; all we had to do was hit the "Commit changes" button. With Git and the command line, there are no buttons or fancy things to interact with. We instead have to tell Git exactly what we want to do.

Git Procedure 4.6 Trying `git commit`. Let's learn how to commit our files using Git. Note that you must have staged at least one file before moving on to this step. See Git Procedure 4.5, p. 51 if you have not done this.

The basic command to commit files is (conveniently) `git commit`; `git` for initiating Git and `commit` to tell Git we are going to be committing files. However, the command also requires a **commit message** (which we also did in Section 3.3, p. 37). This is done with the `-m` switch.

(a) Take a minute to think about the edits you have just made. In a few words, how would you explain to future you and others what you changed? If you could summarize your changes in a sentence or less, how would you? Whatever you decide on will be your commit message.

Do keep in mind the guidelines expressed in A Note On Commit Messages, p. 42. They apply here, too! (And any other time you are committing.)

(b) Type `git commit -m "<your-commit-message>"` into the command line. For our working example, I might type `git commit -m "add birth country"`.

```
git commit -m "add birth country"
[country 137b0bc] add birth country
 1 file changed, 4 insertions(+)
```

Note that the message is surrounded by quotation marks. This is required so that Git doesn't think that each word in your message is a separate command. They can be either single or double quotes as long as the two match.

Also take a look at the output. This tells you the name of your branch, a unique identifier for your commit (so yours will be different than mine), your commit message, how many files were changed, and how many lines were inserted or deleted. This can be useful to verify that the commit worked as you expected.

(c) Just for good measure, try `git status` as an extra verification step (Git Procedure 4.4, p. 51).

It is worth mentioning a few notes about commits since they can be forgotten or confused.

1. A commit is similar to saving a Word document and closing Word completely. Sure, you can go back and reverse your changes or keep editing, but it would take some time to start Word back up and open the file. Same with commits. It is possible to reverse a commit, but it can be annoying to have to do so. Git gives you multiple stages in the finalizing process to help you catch any edits you forgot to make.

2. If for whatever reason you find that you made a commit too early or did so accidentally, you can reset your current branch back to what it was before the commit, according to Git's knowledge. Details can be found in Git Procedure C.2, p. 92 in Appendix C, p. 91.

3. It is possible to have multiple commits in one push. See Subsection 4.3.4, p. 53 for more details.

4.3.4 Step 3: Pushing Files

Take a minute and go back to your GitHub repo for your `aboutme` files. Do the files there reflect your changes of an added country?

Nope, nothing should have changed. Git has, in its three-step process, built in saving graces to help us in case we make a mistake. If `git commit` sent our files right to GitHub, it would be much more difficult to reverse things or correct our mistakes.

Enter the third step in the process: the `git push`. This step is new; we haven't seen it before. When we worked on GitHub, a commit *did* did send our changes to GitHub. But now that we are working locally, an extra step is needed to tell Git that we are truly finished editing *and* that we want our changes to show up on GitHub for the world to see.

In order to understand the syntax of the `git push` command, we need to talk about remotes. A **remote** is a connection between a local repo and a GitHub repo. Remotes are also dependent on the branch you are on. By default, when you clone a repo, Git creates a remote from your local main branch to the main branch of the repo on GitHub. It's a connection; when you tell Git to push changes to GitHub, it knows where to send the changes because of the remote.

There are two types of remotes: **origin** and **upstream**. Upstream remotes are not typically necessary when you are working with yourself and on your own repo. We will come back to them in Subsection 5.4.2, p. 68. To Git, "origin" is the repo that you have cloned. Throughout this section, we have been working with a repo titled `aboutme` that is located on GitHub. This GitHub version of the repo is "origin". When we cloned, we made a copy of "origin" onto our computer and Git established a remote connection from the local computer to GitHub.

This is necessary to understand `git push`. The command has four parts:

1. `git`, telling our computer to use Git

2. `push`, telling Git that we are going to be pushing our committed files somewhere

3. The remote you want to push to. Here, we will use `origin`.

4. The branch you want to push to.

With this information, try Git Procedure 4.7, p. 53 which will guide you through the process and some common errors should you forget one of the four parts.

Git Procedure 4.7 Trying `git push`. This procedure will start with telling you to do the wrong things so you gain an understanding of what to expect if you type the `push` command

incorrectly. The second part will show you the recommended way to push your files back to GitHub.

(a) Let's start with some errors that Git can throw at us.

  (i) What might happen if you forgot to specify the remote and branch? Let's find out: type `git push` into the command line.

  (ii) What if you just forgot to specify the branch? Try `git push origin`.

  (iii) What if you remembered the branch, but not the remote? Try `git push country`

(b) Now let's see how to properly push files back to GitHub. This part assumes you have already staged and committed all of the files you intend to push.

  (i) Take a minute to think about where you are pushing to. Origin? Upstream (if applicable)? What is your current branch name?

  (ii) Once you have decided, fill in the correct pieces of the `git push` formula.

At this point, you are ready to head back to GitHub and submit a pull request!

As mentioned earlier, it is possible to have multiple commits per push. An example of such a situation might be typo correction. Suppose I just finished a long novel and now I'm going back to correct all of my typos. So I create a branch called `typos` and start editing my files to remove my typos. Instead of fixing all of my typos, then staging all of my files, then doing one commit and one push, I could break things up. I could fix all the typos in Chapter 1, then add the files and commit (with a message such as "correct typos ch1"). Then fix typos in Chapter 2 and add and commit. When I'm done, I do one push and Git will push all of the commits to GitHub at one time. Breaking things up like this allows for better version control. More commits leads to more stamps on the version timeline. It also can give you peace of mind. If something horrible happens and I lost all of my book's files, then I only lose the changes for the chapter I'm currenlty working on whereas if I had been editing all at once, I would have lost *all* of my changes for all of my chapters.

This section deserves a final summary.

Git Procedure 4.8 The Three-Step Process. When you are finished editing your files and are ready to send things back to GitHub, follow the Three-Step Process. Don't forget to use `git status` throughout this process!

  1. Stage your files: `git add <files>` (Git Procedure 4.5, p. 51)

  2. Commit your staged files: `git commit -m "<commit-message>"` (Git Procedure 4.6, p. 52)

  3. Push your committed files: `git push <remote> <branch>` (Git Procedure 4.7, p. 53)

When we push to GitHub, your files are pushed to the branch you specified. If the branch had not existed before the push, then Git will create the branch for GitHub, but will not merge it with the main branch. To do so, we need to go back to GitHub and create a pull request. The next section revists the process of creating a pull request and builds on the ideas gained in Subsection 3.3.4, p. 42.

4.4 The Final Steps

We're almost done! At this point, you should have completed the three-step process as outlined in Section 4.3, p. 50. What could be known as the fourth step is to head back to

GitHub and submit a pull request. Again, it does seem silly to ask yourself for permission to include your own changes, but it makes more sense when contributing to someone elses repository. In order for your changes to be included into any main branch, you need to ask the repo owner/moderator(s) for permission to pull in your changes.

It is important to make a distinction between a **push** and a **pull**. When you push changes, you are sending your changes from place to another, such as how we pushed changes from local to origin. Submitting a pull request is equivalent to pushing your changes to the main branch, but since this often requires permission, we typically use the term **pull request**. In this case, you are asking someone else to pull in your changes to their main branch.

Activity 4.9 Creating a Pull Request. A lot of this section will look familiar to the ideas expressed in Section 3.3, p. 37. However, it is worth going over again just to make sure everything stuck.

Note that it is possible to create a pull request using the terminal. However, I prefer to do so on GitHub. The interface is nicer and human-friendly which allows me to be sure that I am doing the right thing and to verify once more that I edited the correct files.

(a) Navigate back to your GitHub repository.

(b) Once there, you might see that a nice bar has popped up at the top of the page with a green button that says you can "Compare and pull request". If you see this, great! Click on the green button and proceed to Task 4.9.d, p. 55. If not, read on.

(c) If your repository looks like nothing changed, never fear. There are a couple of ways to get to where we need to get. Either of the following will work so pick what you like best.

  (a) When in the Code tab, click on the dropdown menu that currently has "main" selected. Click on your new branch title ("country"). A new box should pop up saying that this branch is 1 commit ahead of main. On the right of that box, click "contribute" and then "Open pull request".

  (b) When in the Code tab, click on the branch icon that says "2 branches" (this is different from before…previously it only said one branch). This gives a list of all the branches. Find the "country" branch and click on "New pull request"

  (c) Navigate to the Pull requests tab. Click on the green button near the top right that says "New pull request". Select the branch you want to pull into main from either the "Compare" drop-down menu at the top or the example comparisons table. Click "Create pull request".

(d) Regardless of the method you chose in Task 4.9.c, p. 55 or if you had the automatic pop-up banner, you should be on the same screen. This might look familiar. Notice that your commit message you chose back in Section 4.3, p. 50 appears here. You have the choice to add a further description.

Note the drop-down menus at the top of the page. With these, you can easily decide which branches you want to merge into and which branches you are merging from. You should only see two choices since we only have two branches but this will not always be the case (especially if you are merging from an origin to an upstream, see Section 6.4, p. 75).

You can also scroll down and see all of the files that were changed and all of the changes. This is good to check that everything is as you expected. This also works to check that someone who is trying to contribute to your repo is not destroying your project.

(e) Once everything looks good, click on "Create pull request".

All that's left to do now is to merge that pull request into your main branch. (Note that if you did not own the repository that you just submitted a pull request to, you will not be able to do this step.) Click on "Merge pull reqeust" and then "Confirm merge". Unless you plan on using the branch again, click on "Delete branch" so as to not clutter your branch list. Go back to your Code tab and make sure that the README has been updated with your additions.

Unfortunately, we're not quite done. Now we have another problem: GitHub has changes that our local computer does not! We just merged the country branch into the main branch but our local computer doesn't know that. We have to tell it to **pull** in the changes from the origin remote.

Git Procedure 4.10 Trying `git pull`. The easiest way to bring in new changes from GitHub is to use the `git pull` command. `git pull` works exactly like `git push`. You need to tell Git which remote you are pulling from and which branch.

(a) Head back to your command line. If you need to, navigate (`cd`) to your repo. You should notice that you are still on your `country` branch.

(b) Switch back to your main branch (Remember how?)

(c) Just like with `git push`, decide on your remote and your branch. These will be the remote and branch from which you are pulling. Type in the correct command.

Note that it is possible to pull from other branches. Suppose you were working on the `country` branch and someone else made changes to just the `country` branch (but the changes hadn't been merged into main yet). You could use `git pull origin country` while on the country branch to update your local branch with the origin changes.

Warning 4.4.0.1 Git Procedure 4.10, p. 56 demonstrates that your local repository and GitHub only communicate with each other when you tell them to. Any changes you make on GitHub will not be reflected on your computer unless you pull. Vice versa, any changes you make on your local computer will not be reflected on GitHub unles you push. Be careful with this. If you change a file locally *and* remotely on GitHub without pulling or pushing you might create a **merge conflict**. These are best avoided. The simplest conflicts can be resolved somewhat easily but they can quickly get very complicated. See Section C.7, p. 96 for more information on this.

When working by yourself, merge conflicts are easier to avoid. Just make sure that you only edit at one place at a time (e.g. locally or remotely) then make sure both locations are up to date before editing in another place.

Small note for advanced users: as explained in Section C.7, p. 96, you can edit in both locations at once *as long as you do not edit the same line.* Merge conflicts only arise when the same line(s) has been changed. So if I change lines 3 and 4 locally and 5 and 6 remotely, I can merge, pull, and push without any problems. But if I change line 7 in both locations … eek!

And that's that! You now have made new changes and updated GitHub and your local computer and GitHub repo are on the same page. Great!

This might seem like a lot of work for one tiny edit. And you're right. We edited four lines and it probably would have been easier to use GitHub for that edit. But it's always nice to start simple. But as it turns out, the steps discussed in this chapter are the same for each time you want to update GitHub with changes from local. The next section puts your skills to the test with an extensive activity and also provides a (hopefully) useful summary of the Git process.

4.5 Summary

While the project in Section 4.6, p. 57 is a test of your abilities, it is not a "traditional" test. I don't expect you to have memorized all of the content of the previous chapter. That will come with time. The process of cloning a repository and editing, staging, committing, and pushing files is something you will repeat over and over again as you work through projects. For right now, please go back to the previous sections if you forget how to do something! For your convenience, a summary of the Git process is provided here. Utilize this summary for the project and for your future endeavors. There is no shame in looking something up to make sure you're doing it right, espeically with Git.

What follows is an ordered list of the steps in the Git process when working alone.

1. Create a repository on GitHub. (You can also do this with Git, see Karl Broman's guide[50]. I prefer starting with GitHub, however.) See Subsection 3.2.1, p. 31.

2. Clone the repo to your local computer with `git clone`. See Subsection 4.1.1, p. 46 and Git Procedure 4.2, p. 47.

3. Create an appropriately-named branch with `git branch` and `git switch` or with `git switch -c`. See Subsection 4.1.2, p. 47 and Git Procedure 4.3, p. 48.

4. Edit/create/delete files as you please. See Section 4.2, p. 49

5. Follow the three step process as described in Section 4.3, p. 50. Don't forget about the usefulness of `git status` throughout (Subsection 4.3.1, p. 50)!

   (a) Stage your files with `git add`. See Subsection 4.3.2, p. 51 and Git Procedure 4.5, p. 51.

   (b) Commit your files with `git commit -m`. See Subsection 4.3.3, p. 52 and Git Procedure 4.6, p. 52.

   (c) Push your files back to GitHub (on your current branch) with `git push origin`. See Subsection 4.3.4, p. 53 and Git Procedure 4.7, p. 53.

6. Create a pull request on GitHub and merge your changes into `main`. See Activity 4.9, p. 55.

7. Go back to your local repo, switch to `main` with `git switch main` and pull in your changes with `git pull`. See Git Procedure 4.10, p. 56.

4.6 Working Solo: The Culminating Experience

As mentioned above, this part is a large activity designed to help you practice the skills and ideas learned in the previous sections. It also represents and example of what a typical workflow might look like. You are welcome and encouraged to refer back to the summary and the sections themselves.

The activity below is broken up into multiple parts. In order to help you remember the procedures, I will not be adding in cross references to the corresponding chapters. That being said, I will add in hints and answers where applicable. Do note that some of the steps have no "correct" answer. This will be noted in the corresponding answer.

Activity 4.11 The goal of this activity will be to create a new repository and write two poems (or paste your favorite poems, with attributions). Good luck and have fun!

---

[50]kbroman.org/github_tutorial/pages/init.html

(a) First, you need to create a new repository. Navigate to GitHub and create a new repository. Name it appropriately for the end goal — you can't change it later. Make sure you initialize with a README file.

(b) Clone your repo on your computer. Navigate to the new repo using your terminal.

   Hint. Make sure you are not "inside" your previous repositories. Nested repositories will not work. So `cd` to the folder in which you want to place your new repo.

(c) You will edit your README file first. Create and switch to a new branch, named appropriately.

(d) Open the README file. Change the default title to a better title. Add a sentence or two explaining what this repo is and why you are creating it. Add any other relevant information you think might help other people looking at your repo would need or like. Save the file.

(e) Check if Git is tracking the README file you just edited.

(f) Stage your file for committing.

(g) Commit your changes. Add an informative commit message.

(h) Push your changes to GitHub.

(i) Go to GitHub and open a pull request. Merge your changes into your main branch. You may delete the old branch.

(j) Go back to your local computer. Switch back to main and pull in your changes.

(k) You will now create your poems. Create and switch to an appropriately-named branch.

(l) Create a new `.md` file called `poem1.md`.

   Hint. In VS Code, you can create a new file with the paper-with-a-plus icon in the "Explorer" panel in the top left.

(m) Open the file and use Markdown to write one original poem or paste one of your favorite poems. Make sure to add a title!

   If you are pasting someone else's poem, use proper attribution. They may include, but is not necessarily limited to, name, collection/book from which you found the poem, date published, page numbers, etc.

(n) Stage your new file. Verify that it is in fact staged.

(o) Commit your new file with a good commit message. Verify that it is committed.

(p) Do not push yet! You will add your second poem in a second commit before you push. (So this push will have two commits at one time.)

   Create a new file titled `poem2.md`. Open it and add your second original poem or paste your second-favorite poem (with attribution, of course!). Stage this file and commit it with a good commit message.

(q) Push your new files to GitHub, open a pull request, and merge your changes into `main`.

(r) Go back to local and pull in your new changes on main.

(s) For extra credit, create a new branch and write/paste a third poem. Follow the same processes as above.

If you'd like, use Section B.6, p. 90 to delete your repository. I won't be using it again in this book and this might be good practice to see what deleting a repo looks like. On the contrary, it might be nice to keep the repository to show other people that you are active on GitHub and are open to learning how things work. You can always write this fact in a statement on your README.

And that's that! Hopefully you're starting to get the hang of this process. If you ever need to practice this again (or need a refresher on what to do), revisit this page. It might take a little bit to become fully comfortable with the process and to remember the order in which to do things. In time, you will get it!

Part III

Working With Others

Chapter 5

GitHub Collaboration

Great! You seem ready to dive further into the Git/GitHub maze. While you might do many projects alone, the real world is full of teams and sub-teams collaborating on tasks. I assume that you have read (or at least understand the concepts of) Part II, p. 27 (Chapter 3, p. 27 and Chapter 4, p. 45) since working with others is very similar to working alone. Many of the same ideas return, such as commits, pushes, and remotes, but there are some new twists and new vocabulary to consider and special considerations to be made.

5.1 Before Beginning, Consider This

One of the great features of GitHub is the promotion of collaboration, adding new things, and improving others' work. If your repo is public, anyone can contribute to it. But as the repo owner, you have certain authorities. No one will be able to change things officially without your permission. The main branch will only be changed if you (or other moderators) approve it. This emphasizes the importance of branches (so you can test others' work) and forks (as explored in Section 5.2, p. 63).

As more people collaborate on a project at the same time, the complexity of the branch and fork system increases. With this, it is important to consider potential conflicts. Especially with larger projects, you could have multiple working on the same area at the same time. If two people decide to change the same part of the project at the same time, which edits win? Who gets the privilege of having their changes incorporated into `main`? Sadly, GitHub and Git cannot figure that out by themselves and create a merge conflict, leaving you, the repo moderator, to settle the dispute.

Throughout this part, I will guide you through the process of editing with a team while providing warnings and lessons from what I have learned. I should say that the extent of my knowledge of this topic is not as developed as others' might be. With Git, I find it better to stick with the things I know how to do (add, commit, push, pull, fork, clone, etc.) and if a problem arises, I research how to fix it. As I learn how to correct Git errors, I will update this part of the book and especially Appendix C, p. 91. Regardless, the focus of this book is on the basics and what you need to know to have a successful workflow. If that's what you need, read on.

5.2 Getting Set Up To Collaborate

The first step to collaborating is to recognize that you usually do not have the permissions to edit others' repositories directly. We might say that you do not have **write access** to

those repositories. You won't be able to edit files on GitHub like we did in Chapter 3, p. 27. So how do you contribute to someone else's repository?

5.2.1 Forking

Recall from Chapter 4, p. 45 that in order to edit files on our computer, we had to clone the respository from GitHub. Essentially, this was a copy of your repository on your computer. In order to edit someone else's files, we need to create a copy of their repository *on GitHub*. Then, we edit our copied files and submit a pull request asking the moderator to incorporate our changes.

This process should seem familiar! But how do we make this copy? You can't clone a repository on GitHub (this is a Git concept), but GitHub does provide the ability to **fork**. Like "commit", "fork" is both a noun, referring to a user's copy of the authority repo, and a verb, referring to the action of making that copy.

You may have noticed that GitHub provides a "Fork" button in the top right corner of a repo's webpage (note that you cannot fork your own repos). See Figure 5.2.1.1, p. 64 for a visual reference then proceed to try your hand at forking.
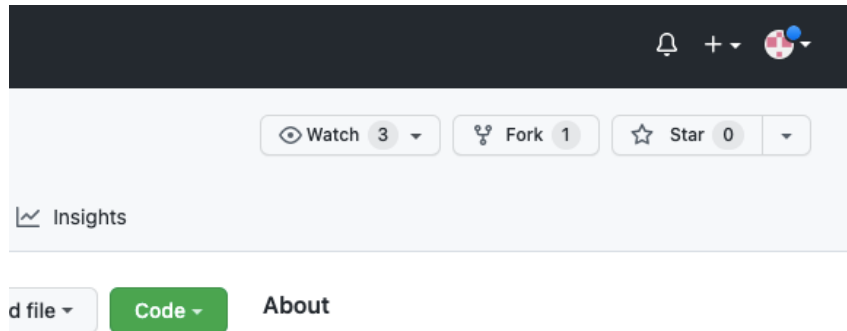
Figure 5.2.1.1 The Fork Button on GitHub

Activity 5.1 Forking On GitHub. This activity focuses on forking a repo and getting a new project started that we will carry throughout this chapter: editing a file on a repository I created.

(a) First, navigate to my `aboutme` repository on GitHub[51].

(b) Locate the "Fork" button in the top right corner.

Note the number next to it. This indicates the number of people who have forked a repo. This also applies to the Watch (people who receive notifications for the repo) and Star buttons. A good sense of how popular and how useful the contents of the repo are can be gained from looking at these numbers (where higher is better.)

(c) Click on the Fork button. A new webpage will appear allowing you to rename your fork and add a description. I recommend keeping the same name as the parent repository, as GitHub suggests. This will make it easier to remember what the repo is for.

That being said, you cannot have two repos with the same name. If you have been following along with the book, you might already have a repository named `aboutme`. In that case, you should choose a different name, but one that is close or similar to the parent repo's.

Click on the Create Fork button.

(d) After a few seconds, your own fork will appear! Take a look at the top left corner where the name of the repo is displayed. You should notice that your personal username and

repo name are there. But underneath that, in smaller font, GitHub tells you the username and repo name from which you forked. This is useful for reminding yourself that you are working on a copy of a repository, not the authoritative one.

Congrats! You've just forked a repository! If you plan on collaborating with others forking will become something you get very used to.

Remember that you just created a copy of my repository. You can make all the changes you want on your copy without asking for permisson from me. But as soon as you try to update my repo, you will have to submit a pull request. A fork is where you edit some things, try some things, and break some things before making a final product of whatever it is you were working on. Then send the changes over to the main repo!

5.3 Editing

This part should seem familiar. Remember how to edit files on GitHub? (Hint: Subsection 3.2.3, p. 34.) Let's try editing a file on my `aboutme` repo. Click on the `aboutyou.md` file on the home page of the repo. This is a Markdown file, just like the README we were working with in Section 3.2, p. 31. Follow Activity 5.2, p. 65 to edit this file. Remember, things should look very similar to Markdown Basics, p. 34 and Checkpoint 3.2.3.4, p. 35.

Activity 5.2   Collaboration Editing: Start With Yourself. When most people collaborate, they make a bunch of changes on their fork. Then, when the changes are ready, they send them over to the main repository. This activity focuses on the first part of that process, editing files on your own fork.

(a) Click on the "edit" button (the pencil icon on the right side of the page)

(b) Follow the instructions on the file to add your own entry.

> To contribute, add your name as a level two heading. This can be just your first name, just your last name, both, all names, initials, a name you like, etc. In a paragraph, type the month, year, and the country and/or state/province you are contributing from. If you feel like it, tell us something cool about you! (See my example.)
>
> Responses will be moderated. No explicit content whatsoever will be accepted (keep things G rated please!).

(c) Once you are done editing, scroll to the bottom of the page, to the Commit changes block. In the box that is autofilled with "Update aboutyou.md", type a more informative commit message. A description is not necessary.

(d) Do not select the button that says "Commit directly to the `main` branch". It is always a good idea to create a branch when editing someone else's work.

Select the button that allows you to create a new branch and type in a better branch name.

(e) Notice that the green button now says "propose changes" rather than "commit changes". Again, this is because you do not have permission to commit changes to my repo, just propose them. Click on this button once you have changed the branch name.

(f) This will conveniently bring you right to the pull request page, which might look familiar. Your previous commit message will automatically be there. The only thing you really need to do is click on "Create pull request". (But read below before you click it.)

---

[51]`github.com/ian-curtis/aboutme`

Also take note at the top of this page GitHub tells you what branches are being merged. You will probably see

    base: main ← compare: <your-branch-name>.

This means that you are taking your branch and attempting to merge your changes into the main branch from the "base" repository, or *your repo.*

Like before, if you scroll down, you can see all of the changes you made. Later, I will be able to see this too, to make sure that you only changed what you were supposed to and didn't write anything R-rated.

Now you can create the pull request.

(g) Here, you are asking yourself for permission to merge into your fork. Hopefully you give yourself permission to do so. Click on "Merge pull request" and then "Confirm merge". You may delete the branch if you'd like.

Assuming you have completed Activity 5.2, p. 65, you might notice that a new box has popped up in the main page of your fork. It might say something like "This branch is 2 commits ahead of `ian-curtis:main`". This is how you create a pull request.

Now you can make as many changes and commits as you want before opening a pull request (this will change the message on the dialog box). You don't even have to merge your changes into your fork's main branch before opening a pull request. However, I usually choose to do so just so I know that all of my changes from my various branches are included where they are supposed to be. Feel free to continue making edits and/or commits to your entry in the `aboutyou.md` file before continuing!

Activity 5.3 Collaboration Editing: Sending Your Changes Away. This activity assumes you have been following along throughout the section. Activity 5.2, p. 65 must be completed before this activity. Just like in Subsection 3.3.4, p. 42, we will use the "Contribute" option on the Code tab of a repo (see Figure 5.3.0.1, p. 66) even though there are other ways to open a pull request.
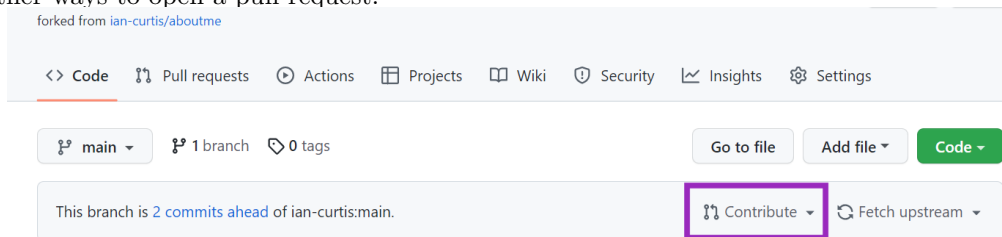
Figure 5.3.0.1 The Contribute dialog box on GitHub.

(a) Make sure you are on your fork's home page (the Code tab). Click on the Contribute drop down menu and then the green Open pull request button. This will bring you to a new page containing familiar elements.

(b) This task is all about noticing. Take a minute to identify each of the following elements on the page. These will help you in the future to recognize that you are opening a pull request for someone else's repository and can be a reassurance that you are in the right place.

(a) The title of the repository has changed. You are no longer on your fork, but on the forked-from repo. As such, the Watch, Fork, and Star data have changed to describe that authoritative repo.

(b) There is a dialog box indicating which branches are being merged. There are more boxes than before. There should be four, allowing you to specify which

repo and branch to want to merge from and which repo and branch you want to merge to. Usually GitHub gets it right, but not always, so check to make sure these are what you want.

The **base repository** is the repo you are merging to (here, the authoritative repo), the **head repository** is the repo you are merging from (here, your fork). The base branch is the specific branch you are merging to (often `main`) and the compare branch is the branch from your fork you are merging from (often `main` for me personally).

(c) After these boxes is, hopefully, a green checkmark and an "Able to merge" message. This is what you want. You might also get a red message about conflicts or other messages. In this case, you might have a merge conflict and will need to do some extra work. See Section C.7, p. 96.

(d) There is a summary of all the commits made on the branch (with their commit messages)

(e) There is info on all of your specific changes in each file

(c) Once you have found all of the above, go ahead and click on the "Create pull request" button. If your pull request has more than one commit, you will have to enter a title for the request. This is where a description might be helpful. Choose a title that describes what you are doing. If you would like to supply more information, ask a question, or give clarification, type as much as you want into the "Leave a comment" text box. Then click "Create pull request".

(d) This time notice that there is no option to "Merge" into the main branch. This is because you do not have write access to the parent repository. At this point, wait for a moderator from the authoritative repo to approve your pull request.

Note that forking can drastically increase the complexity of the GitHub tree. Recall from Figure 3.3.4.1, p. 44 that branches make another branch on the tree which you eventually merge back into the main branch, or the trunk of the tree. Cloning made this more complex due to the copy on your computer. Forking adds another copy into the mix. Figure 5.3.0.2, p. 67 provides a possible diagram of the forking process. But there are many different variations to this diagram as it is up to you how many branches/commits to make and to which branch you push.
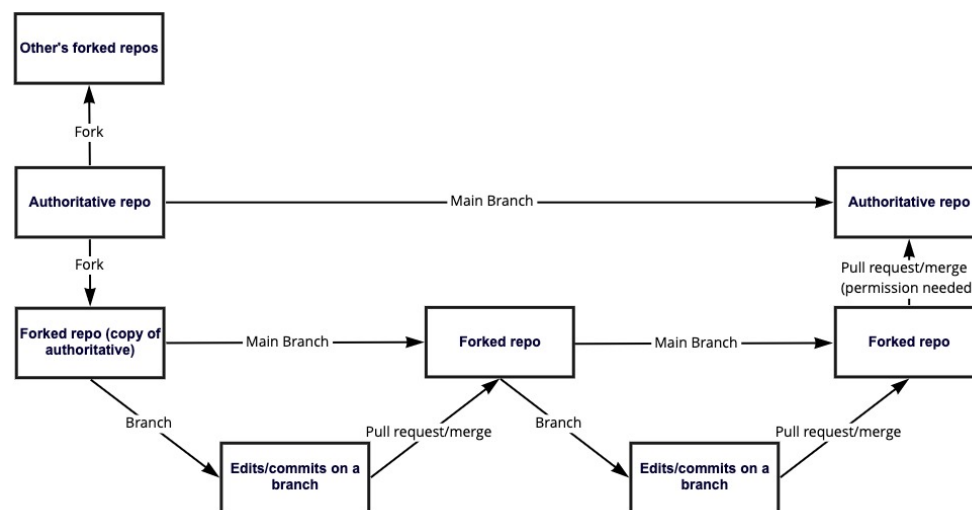

Figure 5.3.0.2 Example of GitHub Tree with a fork

Now you can sit back and relax until a moderator reviews your contribution. If they

make a comment or approve your pull request, you will get an email to the email address on your account. Once they approve your work, great! Congrats! Are you done? Proceed to Section 5.4, p. 68 to find out!

5.4 Fetching Upstream

5.4.1 First, Fetching

To answer the question in Section 5.3, p. 65, we are not quite done. See, even though your fork and the parent repository are connected, updates in one are not automatically reflected in the other. So if you just contributed to a repository and the pull request was accepted, you should be fine. But if someone else's pull request also got approved, your fork will not have those changes. You have to tell GitHub to grab those changes and update your fork.

The act of grabbing changes from the parent repository is called **fetching**, aptly named since that is exactly what you are doing: fetching changes. Technically this is distinct from a related **merge**. First, you fetch the changes, then you merge the changes into your fork.

You might remember using `git pull` to fetch in changes. As we will see later, this essentially is the same thing. Pulling fetches and merges at the same time.

Before continuing, you must wait until I accept your pull request from Section 5.3, p. 65.

5.4.2 Second, Upstream

Another important concept is that of **upstream**. Remember from Subsection 4.3.4, p. 53 when we talked about remotes? Back there, I mentioned that there were two types of remotes: origin and upstream and that we would talk about upstream later. Well, now it's later! This idea will become important again when we use Git to collaborate (see if you can figure out why).

Any repository you own will have the remote identifier of "origin", for you. Any repo you do not own *and that you fork* will have the remote identifier of "upstream". Remember that a remote is like a connection between repositories; it's the definition of the link between them. When you fork a repo, you are creating a remote connection between the parent repo (upstream) and your newly-created fork (origin). As we will explore in Chapter 6, p. 71, when you clone a repo, you are create a remote connection between your repo (origin), which would be a fork, and your personal computer (local).

5.4.3 Third, Fetching Upstream

So in order to have changes from the parent repository reflected in your own fork, you have to **fetch upstream**. GitHub makes this easy! Remember that dialog box from Activity 5.3, p. 66? Back then, we used the "Contribute" button to start a pull request. Here, we will use the option right next to it: "Fetch upstream".

First, though, go back to the homepage of your fork. You might notice that a slightly different box has appeared. Now it might say something like "This branch is 1 commit *behind* `ian-curtis:main`" (you could have a different number). Earlier you were *ahead* of `ian-curtis:main`. What's going on here?

Ahead vs. Behind.

GitHub uses **ahead** and **behind** to tell you how your fork currently sits compared to the repo you forked from. Any commit that you have made to your fork that has not been merged into the parent repo ticks up the number of commits ahead you are. Any commit that has been made to the parent repo that is not reflected in your fork ticks up the number of commits behind you are.

As an example, you might have a message like "This branch is 2 commits ahead, 3 commits behind `<parent-repo>:main`". This means you have made 2 commits that are not on the parent repo and that the parent repo has 3 commits that you have not fetched.

Warning 5.4.3.1 Fetch Upstream Often. In my opinion, you want to keep the least amount of commits behind. If you are working on someone else's work, you want to stay up to date. If some big change happens, you should make sure you have that change on your fork. Why? Well, you don't want to overwrite anything someone else just did. You also don't want to work on adding a new feature that is already incorporated. Maybe the new change helps you do your work better/faster. You also want to avoid potential merge conflicts because someone else changed the same files you are working on.

Therefore, it is usually recommended practice to fetch upstream as often as you can. At the very least, daily, especially if the project is popular and updated frequently. It is possible that there is no new upstream content to fetch but hey, at least you checked!

This warning will repeat in Chapter 6, p. 71 when we talk about pulling and pushing.

You may be wondering why you are even any commits behind! If your addition to my repository was accepted, then our repos should match, right? Not quite. Actually, that act of accepting your pull request and merging into my main branch was a commit in itself which is where the mystery commit comes from.

Activity 5.4 Fetching Upstream. Enough talk, let's fetch that upstream!

 (a) Find the "Fetch upstream" dialog box on the home page of your fork.

 (b) Notice that GitHub will fetch and merge the upstream changes in one step (even though it is technically two steps). Great!

   It also emphasizes Warning 5.4.3.1, p. 69 through a somewhat passive-aggressive reminder: "Keep your fork up-to-date with the upstream repository".

   To make the magic happen, click on "Fetch and merge".

 (c) Bam. Done. But do make sure your fork now says

   This branch is up to date with `ian-curtis/aboutme:main`

   (in general, this would be the name of whatever repo you forked from). Yours may also say, depending on when you fetched, that your branch is a number of commits ahead (but none behind).

And there you go! Now you are ready to keep on editing or to go about your day. If you are interested in some other GitHub features related to collaborating, continue to Appendix B, p. 87. Otherwise, go to Chapter 6, p. 71 to see how to use Git commands and local edits to collaborate.

Chapter 6

Git Collaboration

This chapter is closely related to Chapter 4, p. 45 and Chapter 5, p. 63 in that the concepts there carry over and are combined together here. The workflow and steps described in this chapter are most similar to the typical day-to-day work from many people around the world when it comes to Git and GitHub.

6.1 Getting Set Up: A Few Extra Steps

Before you are able to jump right in and start editing files locally, there are some things that have to happen first. Like in Section 5.2, p. 63, we need to fork the repository you would like to edit. Although it is possible to use Git to edit on a repository that has not been forked, you will find problems when you try to push your changes since you do not have write access to the repo.

So, to continue on with this section, make sure you have my `aboutme` repository forked. Recall that we did this in Subsection 5.2.1, p. 64. If you've done this already, you do not need to do it again. If you haven't, review the instructions in Subsection 5.2.1, p. 64.

I would also like to repeat something I said back in Section 4.1, p. 45:

> It's not always feasible to edit file on GitHub, as we did in Section 3.2, p. 31 and Section 3.3, p. 37. Certain text editors (for instance, Visual Studio Code which you might also have installed) have features built in to ease coding and file editing. VS Code extensions might add color to text to help you see what's going on in your code. Others provide bug fixing and error catching before the errors actually occur. In general, you will edit files on your personal computer and will want to share them with the world. *I would stay away from editing on GitHub unless you have a small correction and/or you don't need any special editing tools.*

The rest of this section and chapter will look a lot like Section 4.1, p. 45 and Chapter 4, p. 45 with a few changes. With that being said, do you remember what the first step to editing locally was?

6.1.1 Cloning A Fork

Hopefully you didn't peek ahead! The first step is to clone our repo. Remember that in order to edit files locally, we need a copy of the repository on our computer and the way to do that is to clone.

This again increases the complexity of the Git tree. From an authoritative repo we made a copy on GitHub, called a fork. We did this so that we could have all freedom to edit the files as we pleased without breaking any of the official files. From there, we need to clone to make a copy of the fork on our computer. Do make sure you are cloning your fork, *not the repo you forked.*

Checkpoint 6.1.1.1 So go ahead, use your terminal clone your fork of my `aboutme` repository. Before you do so, make sure that your branch is up to date with `ian-curtis/aboutme:main`.

Don't remember how to do this? Git Procedure 4.2, p. 47 might be useful, as may be the solution to this exercise.

Great! Your repo should now be cloned on to your computer. Carry on.

6.1.2 Don't Forget About Branches!

For nearly every edit we've made in this book, we've made a corresponding branch. This is no different. I hope you understand the importance of branches, even if you are working with yourself.

Since my repository is public, anyone can edit the files there. Therefore, I will approve pull requests from anyone as I will assume they are working through this book. However, the task in this chapter will be exclusive to just those who are reading this book. There is file called `secret.txt` with no instructions or anything at all. Later on, you will be adding a line to this text file.

For now, create a good branch name for this edit, perhaps `secret`? Remember that cloning a repo does not navigate inside of the repo so make sure you use `cd <your-repo-name>` before creating a branch.

Checkpoint 6.1.2.1 Make a new branch. Switch to that branch. See Subsection 4.1.2, p. 47 if you need a refresher.
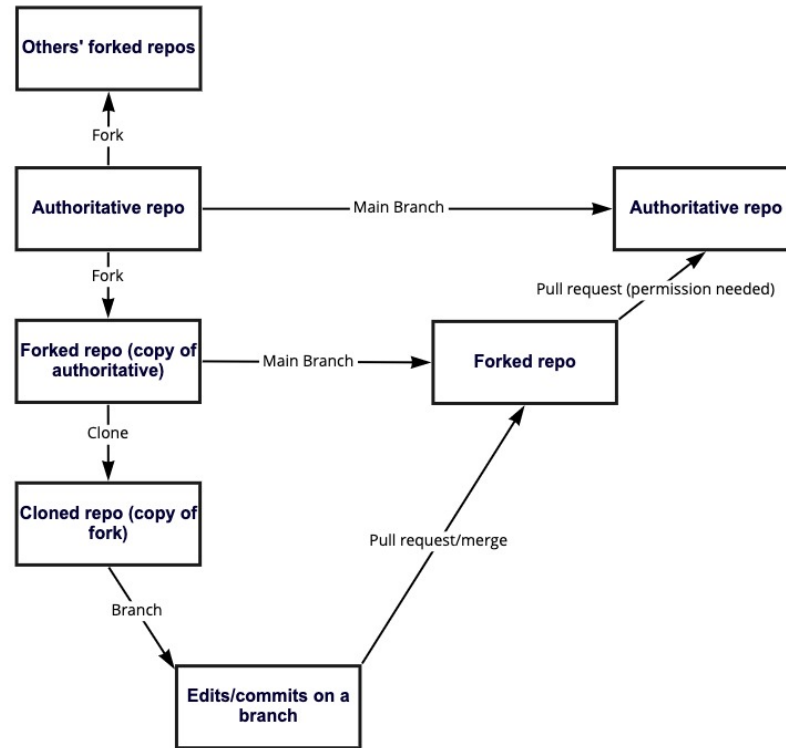
Figure 6.1.2.2 Diagram of a fork, a clone, and a branch

Take a look at Figure 6.1.2.2, p. 73 for a visualization of what the Git tree might look like at the end of the editing process. This is the end goal. Then, let's get to editing.

6.2 Collaboration Editing With Git

As before, the title is a little misleading. Git does not offer editing features, but, once again, it does keep track of your edits and commits for version control.

This section will be very short. All you need to do is open `secret.txt` and type one line of whatever you want (G-rated of course). Literally could be anything. Just something random. Maybe the following will inspire you:

- "Tick tock" went the tick tock croc.

- Have you ever tried ham and peanut butter?

- The jalapeño, who was too hot, walked into the ocean to cool off and was eaten by a shark who became the first creature to breath fire underwater.

- I recently tripped up the stairs.

Whatever you choose, try to stick to one line. When you are done, save the file (which will tell Git that there are active changes) and move on. The next steps are similar, but not exactly the same as those in Section 4.3, p. 50.

6.3 Revisiting the Three Step Process

Just as in Section 4.3, p. 50, this is a very important section. Take care to do everything in the correct order and with the correct keywords.

Recall that the three step process for sending changes back to GitHub consists of

1. Staging your files for sending

2. Commiting your changes

3. Pushing your changes to GitHub

Since we've already covered much of this material, I won't go into as much detail. I will, however, point out any significant differences. As you move forward, don't forget about the handy `git status` command to remind you what files have been edited, staged, or committed.

6.3.1 Staging Files

This step will not be any different from before. Use Git Procedure 4.5, p. 51 to stage your edited `secret.txt` file.

Remember that `git add` is pretty flexible. If you've edited more than one file, you can choose which files to add. Maybe you want to make two commits. In that case, stage some files and commit them, then stage the other files and commit them.

6.3.2 Committing Files

This step will also not be any different from before. Just remember that you should try to design your commit messages to be short and descriptive. If the repo you are contributing to has a preference for how commit messages are written, you should try to follow that as best you can. Otherwise, don't be offended when your commit message is changed.

Git Procedure 4.6, p. 52 will probably be helpful as will the reminders at the end of Subsection 4.3.3, p. 52. Commit your `secret.txt` file and provide an apt commit message (especially one that shows you were reading this book).

6.3.3 Pushing Files

Alright, here is where some small changes come up. In Section 4.3, p. 50, we pushed to the origin repository. But from Section 5.2, p. 63 and Section 5.4, p. 68, when you fork, you introduce another remote: upstream. And now that we have a clone, we also have a local repo.

Alright, so what about right now? You have your changes staged and committed, but to where do you push? Local? That won't work since you already have the changes locally. Upstream? It might make sense to push directly to the authoritative repo, but since you don't have write access, this will likely not work. The right thing to do here is push to origin. Remember, "origin" in this case, refers to your fork.

From Subsection 4.3.4, p. 53 (and Git Procedure 4.7, p. 53) can you figure out what to type into the terminal to push to origin?

Remember you need to specify the remote and the branch you are pushing to so `git push origin <branch-name>` will do nicely.

Believe it or not, Git Procedure 4.8, p. 54 was written in generic terms, so it applies here as well! Keep it handy, or perhaps keep Section 4.5, p. 57 and Section 6.5, p. 76 in mind as you proceed in your Git career. Now let's go create a pull request from that push we just made.

6.4 The Final Steps

All that's left to do is to create some pull requests. There are technically two ways you could make this work. You can either open a pull request from your new branch directly to the authoritative main branch or you can merge from your new branch to your fork's main branch and then to the authoritative main branch.

6.4.1 Opening a Pull Request

Use Activity 4.9, p. 55 to go back to GitHub and start opening a pull request. When you arrive at the pull request creation screen (titled "Comparing changes"), you generally have two choices, as mentioned above. The following list describes the branches and repos you should have selected at the top of the screen, under the page heading.

1. *Merge from your new branch to your fork, then to the authoritative.*

   Your base repository should be `<your-username>/<your-fork>`, base should be `main`, and compare should be `<your-branch`. Do note that once you select your fork for the base repository, the page will reload and the base and head repository options will disappear.

   Then, click Create pull request and merge in to main. You may now open another pull request with options base repository of `<authoritative-username>/<authoritative-repo>`, base should be `main`, head repository should be `<your-username><your-fork>` and compare should be `main`.

2. *Merge from your new branch directly to the authoritative.*

   Your base repository should be `<authoritative-username>/<authoritative-repo>`, base should be `main`, head repository should be `<your-username><your-fork>` and compare should be `<your-branch>`.

   After either of those two, click on "Create pull request", enter a title, and click "Create pull request" again. From there, you will have to wait for approval and moderator merge.

6.4.2 Pulling/Fetching Upstream

Recall from Section 5.4, p. 68 that you can update your fork with the latest upstream changes through the "Fetch upstream" button. That works great for updating the fork, but it doesn't update your local clone. There are two main ways in which you can update both your fork and for clone with new changes to upstream.

Git Procedure 6.1 Updating Origin and Local, Method 1. This method will first fetch upstream on GitHub, then pull in changes from origin.

(a) Use Subsection 5.4.3, p. 68 to fetch (and merge) upstream content into the main branch of your fork (origin).

(b) Now, head back to your terminal and make sure you are navigated to your clone. Do a check to make sure you are on your main branch; if not, use `git switch main`.

   If you are planning on making more changes to a branch, but want it updated to the

latest upstream changes, you can stay on a branch. If you have any uncommitted changes, you might make a merge conflict.

(c) Now use `git pull` to pull in the changes from your recently-updated fork. Git Procedure 4.10, p. 56 may be helpful. Just remember that you are pulling in changes from the main branch on origin.

Git Procedure 6.2 Updating Origin and Local, Method 2. This method will first set up a remote to upstream, pull in changes from upstream, then push changes to origin.

(a) Head to your terminal and make sure you are navigated to your clone. Do a check to make sure you are on your main branch; if not, use `git switch main`.

If you are planning on making more changes to a branch, but want it updated to the latest upstream changes, you can stay on a branch. If you have any uncommitted changes, you might make a merge conflict.

(b) This step will only have to be done once per repo that you have forked, then cloned. When you clone a fork, the connection between origin and upstream is not carried over. If I tried to pull from upstream without this connection, I would get an error:

```
fatal: 'upstream' does not appear to be a git repository
fatal: Coult not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

This is troublesome! Clearly upstream exists; how else would we have forked it? To fix this, you need to manually set the remote connection. Again, this only has to be done once per forked/cloned repo.

  (i) Navigate to the authoritative repo's homepage and copy its HTTPS clone link, just as if you were going to clone the repo.
  (ii) Head back to your terminal and use `git remote add upstream <https-link>`. There will be no output.
  (iii) Verify that it worked with `git remote -v`. This will print out the HTTPS links for origin and upstream. Make sure your username is on the origin remotes and the official repo/username is on the upstream remotes.

(c) Ok, that was the hardest part. Once again, that only has to be done once. After that, every time you want to pull from upstream and update local and origin, you just need two commands: `git pull upstream main` and `git push origin main`.

`git pull` will fetch and merge upstream changes from its main branch and `git push` will take all of your files and push them to origin, updating them there.

Warning 6.4.2.1 Use `git pull` Frequently. On a repository for which you are an active editor, do not get behind on your pulls. As mentioned in Warning 5.4.3.1, p. 69, sometimes changes to upstream can improve or change how you work locally. Pulling often keeps your work at the very edge of the Git tree and reduces the likelihood you create a merge conflict. You also want to push back to origin often to prevent merge conflicts with yourself.

6.5 Summary

This section is a complete summary of the Git/GitHub process. In some cases, there is more than one way to do a step. I try to provide all methods, but truthfully, I don't use

certain methods. I've found a workflow that works for me and I've done it so much that the commands are second nature now. You might think about saving this section for future reference.

Some of these pieces are very similar to the summary in Section 4.5, p. 57, but is designed for collaborative work (where you are editing someone else's repository). You are welcome to follow the instructions in Section 4.5, p. 57 and have others edit your files, too.

1. Navigate to a repository you would like to edit and fork that repository. See Activity 5.1, p. 64 in Subsection 5.2.1, p. 64.

2. Use `git clone` to clone your fork (not the repo you forked from) on your personal computer. See Git Procedure 4.2, p. 47.

3. Create a good branch name (make sure you switch to that branch after you make it). See Subsection 4.1.2, p. 47 and Subsection 6.1.2, p. 72.

4. Edit those files! (Section 6.2, p. 73)

5. Follow the three step process as described in Section 6.3, p. 74. Don't forget about the usefulness of `git status` throughout (Subsection 4.3.1, p. 50)!

   (a) Stage your files with `git add`. See Subsection 4.3.2, p. 51 and Git Procedure 4.5, p. 51.

   (b) Commit your files with `git commit -m`. See Subsection 4.3.3, p. 52 and Git Procedure 4.6, p. 52.

   (c) Push your files back to your fork on GitHub (on your current branch) with `git push origin`. See Subsection 4.3.4, p. 53, Subsection 6.3.3, p. 74 and Git Procedure 4.7, p. 53.

6. Create a pull request to merge your changes into your fork's main branch.

   Or create a pull request to merge your changes from your fork's branch directly to the authoritative repo's main branch. See Subsection 6.4.1, p. 75

7. Wait for your changes to be merged into the main branch.

8. After your changes have been merged, pull the changes into your local clone and your fork with `git pull upstream main` and then `git push origin main`. Remember to do this often even if you don't have any pending changes to be merged. See Subsection 6.4.2, p. 75.

   You may need to set an upstream with `git remote add upstream <link>` (Git Procedure 6.2, p. 76)

6.6 Working Together: The Culminating Experience

As in Section 4.6, p. 57, this part is a large activity designed to help you practice the skills and ideas learned in the previous sections. It also represents an example of what a typical workflow might look like. You are welcome and encouraged to refer back to the summary and the sections themselves.

As before, the activity below is broken up into multiple parts. In order to help you remember the procedures, I will not be adding in cross references to the corresponding chapters. That being said, I will add in hints and answers where applicable. Do note that some of the steps have no "correct" answer. This will be noted in the corresponding answer.

Activity 6.3 This activity will involve you contributing to the repository for this book. This will help me get an estimate of how many people are reading the book (since I can see the number of forks). You will be typing a short paragraph of your desired superpower.

(a) Navigate to the GitHub repository for this book[52], `ian-curtis/gitstarted`.

(b) Fork the repository.

(c) Make sure your branch is up to date with `ian-curtis/gitstarted:main`.

(d) Clone your fork on to your computer. Navigate to the repo using your terminal.

(e) Create a new, appropriately-named branch. Switch to that branch.

(f) There is a file called `superpower.md` inside of the `book-activities` folder. Use your terminal to open that file (if using VS Code, use the terminal; if not, try to use your terminal).

(g) Edit the markdown file with your contribution. Instructions are in the file are pasted below:

> Here is the file for the culminating experience of Chapter 6, p. 71 from *Git Started*. Below, type your GitHub username as a level two heading and then type a short paragraph of the superpower you wish you had and why. This could be anything!! See my example.

Make sure to save the file when you are done.

(h) Make sure Git is tracking your file.

(i) Stage your file.

(j) Check to make sure your file got staged correctly.

(k) Commit your file with a good commit message.

(l) Push your changes back to your fork on GitHub

(m) Go back to GitHub and merge your changes into your fork's main branch, then into the upstream main branch.

OR Go back to GitHub and open a pull request to merge your changes directly into the upstream branch.

(n) Wait for me to accept your pull request and merge your changes into the main branch.

(o) In the meantime, switch back to main and set an upstream branch for your clone.

(p) Once I have accepted your changes, make sure your fork and your clone both have those changes.

(q) For extra credit, create a new branch and do some typo correction on this book. If you do more than one chapter/section, try to do multiple commits. Submit a pull request explaining your changes and I might merge it!

Or, learn PreTeXt, the markup language this book is written in[53] and suggest some edits or new content.

(r) For as long as you wish to stay up-to-date on the code for this book, pull from upstream often.

---

[52]`github.com/ian-curtis/gitstarted`
[53]`pretextbook.org`

And that's that! Hopefully you're getting the hang of this process with all of this repetition. If you ever need to practice this again (or need a refresher on what to do), revisit this page. It might take a little bit to become fully comfortable with the process and to remember the order in which to do things. In time, you will get it! Good luck in the rest of your Git/GitHub journey!

Appendices

Appendix A

Customizing the Terminal

Note that I do have a Mac, so the features I use and like will be here. If you are a Linux user and have something to share, submit a pull request or start an issue and I will look into it.

A.1 For Mac Users

Sorry, Windows users, this is only for MacOS users. If you are a Mac user, you will likely appreciate and benefit from these customizations.

A.1.1 A New Terminal: iTerm2

Instead of using the built-in terminal that comes with your computer, consider downloading a new terminal, iTerm2[54]. The default Mac terminal does great but if you are having trouble seeing what's going on and want more colors or wish you didn't have to type so much, then iTerm2 is for you.

This new terminal is free and has a multitude of nice features[55] such as

1. split panes/tabs

2. search

3. fun colors

4. autocomplete

5. copy mode

6. paste history

and many more. I personally don't see any disadvantage to this terminal. Plus, it works with all the popular shells. To download, visit the link above and simply download it. Once it's installed, it works immediately.

---

[54]`iterm2.com/index.html`
[55]`iterm2.com/features.html`

A.1.2 A New Shell: `fish`

Many computers rely on the `bash` shell to do most of their work. Mac computers now use `zsh` as their default shell. Shells have their own unique aspects and differences and frankly, it doesn't really matter which shell you use unless you have a good technical reason to change. Shells are pretty advanced and I honestly don't have much information to provide other than what a Google search can do since any shell will work for the content provided in this book.

That being said, I do recommend the shell `fish`[56]. This shell brings more nice colors, better autocomplete than iTerm2 (it will recommend as you type!) and, in my opinion, organizes the information on your terminal quite nicely.

Installing is not terrible. I assume you used Homebrew to install Git (Git, p. xv in Materials: Mac, p. xv). If you didn't follow the instructions there to get Homebrew installed as it makes `fish` installation easy:

1. type `brew install fish` into your terminal

2. add the shell to your list of approved shells with `sudo sh -c 'echo /opt/homebrew/bin/fish >> /etc/shells'` (for newer Macs) OR `sudo sh -c 'echo /usr/local/bin/fish >> /etc/shells'` (for older Macs).

3. change your shell to fish with `chsh -s /opt/homebrew/bin/fish` (new Macs) or `chsh -s /usr/local/bin/fish` (older Macs)

After restarting your terminal, everything will be ready for use! For further customization of `fish`, type `fish_config` in the terminal which will open up a web interface.

If you don't like `fish`, you can switch back to `bash` or `zsh` with `chsh -s /bin/bash` or `chsh -s /bin/zsh`.

A.2 For Windows Users (∗)

TBA

A.3 For Linux Users (∗)

I don't have much experience with Linux distributions but a quick search online has informed me of the following information. You are responsible for determining the exact steps needed to implement anything.

A.3.1 A New Shell: `fish`

Many computers rely on the `bash` shell to do most of their work. Mac computers now use `zsh` as their default shell. Shells have their own unique aspects and differences and frankly, it doesn't really matter which shell you use unless you have a good technical reason to change. Shells are pretty advanced and I honestly don't have much information to provide other than what a Google search can do since any shell will work for the content provided in this book.

That being said, I do recommend the shell `fish`[57] which is advertised as available on Linux distributions. This shell brings more nice colors, autocomplete (it will recommend

---

[56]fishshell.com/
[57]fishshell.com/

as you type!) and, in my opinion, organizes the information on your terminal quite nicely. Once installed, you should be able to use the command `fish_shell` which will open up a web interface with settings.

A.4 For All

A.4.1 A New Git Interface

Now that you know Git basics and terminology, you might like interfaces that reduce your terminal usage. GitKraken[58] is one of those choices. Although I haven't used it, it does look nice. It is free with a paid option and is available on Windows, Mac, and Linux.

From a simple browse of the website there appears to be a good brach/contribution graph and clear tabs with specific functions. Perhaps worth trying out!

A.4.2 GitHub Desktop

GitHub does offer a desktop interface[59] to help you with Git features and managing collaboration. If you would rather do these things with an app (as opposed to switching from your text editor to a website) it might be worth downloading GitHug Desktop. The website itself is pretty basic, but the GitHub documentation[60] has a good start-up guide and more.

A.4.3 Understanding Version Control: Git Lens

My preferred text editor is Visual Studio Code which is developed my Microsoft. There is an extenstion to VS Code called Git Lens[61] which,

> ...supercharges Git inside VS Code and unlocks untapped knowledge within each repository. It helps you to visualize code authorship at a glance via Git blame annotations and CodeLens, seamlessly navigate and explore Git repositories, gain valuable insights via rich visualizations and powerful comparison commands, and so much more.

I haven't tried this extension out but it seems like it can be extremely helpful for understanding what files have been changed (and where) as well as for determining what files are in each step of the three-step process. It may even be useful for resolving merge conflicts!

---

[58]`gitkraken.com/`
[59]`desktop.github.com/`
[60]`docs.github.com/en/desktop/installing-and-configuring-github-desktop/overview/getting-started-with-github-desktop`
[61]`marketplace.visualstudio.com`

Appendix B

Extra GitHub Content

There are many features available on GitHub and we have barely scratched the surface. This book is not intended to explain the more advanced and niche aspects, but some of what GitHub offers is very useful to a wide audience and is worth a short discussion.

All of the following can be found on repositories regardless of whether they are private or public. As you become more familiar with GitHub, this page may become useful to exploring some of what tools GitHub has to offer.

B.1 The Issues Tab

You've seen when making pull requests that you can add comments or descriptions to help moderators understand what you are doing. This feature is also used as a place for conversation regarding the pull request and the problems is solves or brings up.

But what happens if you want to talk about a problem without a pull request? What if you don't have the time or experience to edit a repository's contents but want to let moderators know of a potential bug or feature request? GitHub has something for you!

The issues tab is designed as a place of conversation between developers, users, and other members of the community. It runs like a forum where anyone can create a thread (including the repo's owner) and anyone can reply or supply advice. Moderators will often tag issues and with a well-designed commit message, can close an issue automatically.

Some issues have a label of "good first issue". This indicates that the problem mentioned in the issue is not very difficult to fix and with a basic understand of that repo's files, a beginner could fix the issue. If you are interested in developing, I would recommend starting with good first issues.

Note that the Issues tab will not appear on forks, only on the authoritative repository.
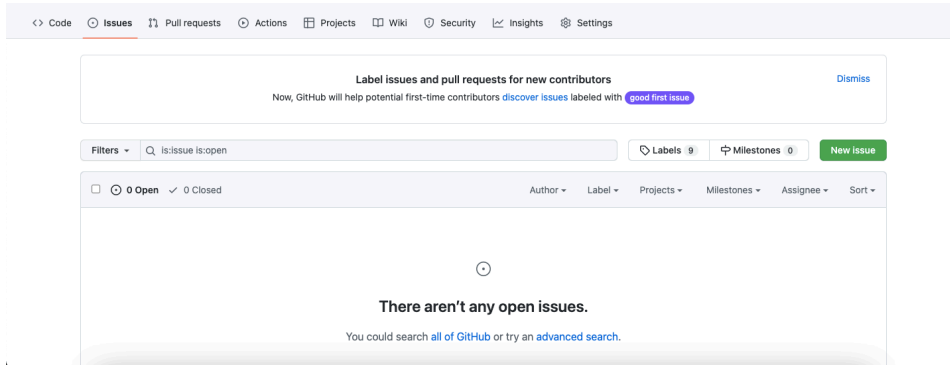
Figure B.1.0.1 Example Issues page for this book
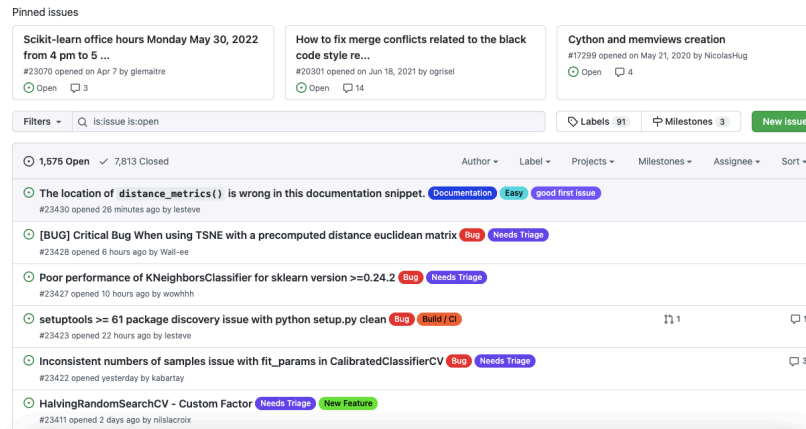
Figure B.1.0.2 Example Issues page for the Python module `scikit-learn`[62]

B.2 Commit History

On the home page (Code tab) for all repositories a commit history can be found, full with contributor usernames, commit messages, and links to further information. Directly above the list of files, you can find information on the most recent commit. At the right side of this area is a clock icon with an arrow circling counterclockwise followed by a number of commits (See Figure B.2.0.1, p. 88). This is the total number of commits to that repository.

Figure B.2.0.1 The number of commits to a repository

Clicking on this will give you more information on how often someone updates a repository and how many people do so. You are also able to see past branch names. Clicking on a specific commit will bring to a page with the files changed and any comments that were made.

Not everyone will be interested in this feature, but it is useful to know it is there should you want to take advantage of it.

---

[62]`github.com/scikit-learn/scikit-learn/issues`

B.3 The About Panel

Most popular repositories have created and worked hard on their about panel. On any repo, this appears on the right side of the home page (Code tab) and it titled "About". This panel may include

- A short description

- A website

- Topics (these are like tags)

- A release version number

- Any packages published in the repository

- Any environments associated with the repository (such as GitHub Pages)

Also on this side of the page is a list of contributors, and the distribution of the langauges of the files in the repo (this can be off since GitHub doesn't recognize some file extensions). It's always nice to glance at this part of the page when looking at a repository just to get a general idea of what's going on or if it's what you are looking for. (A README will also help you understand more specific ideas about the repo content.)

B.4 The Branch Tree Diagram (Network Graph)

Scattered throughout the book are various tree diagrams (created by me) depicting example branch pathways. Of course, these are highly variable depending on how many branches you have, commits you make, and contributors there are.

Turns out, GitHub actually makes a graph of your branches for you, as you create, push, merge, and delete them. This is visible on all public repos. To get to this graph, visit the Insights tab and then the Network page from the left sidebar. And there it is! It might not look very exciting, depending on your workflow. Consider the one for this book as of May 2022 in Figure B.4.0.1, p. 89.
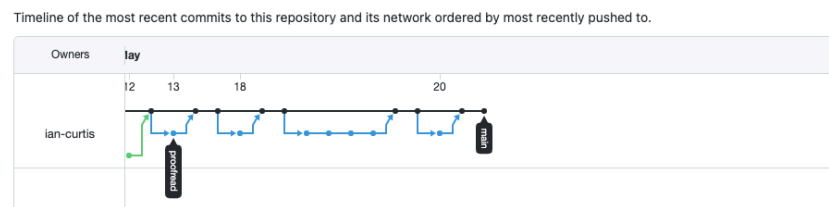
Figure B.4.0.1 Network graph for this book (as of May 2022)

B.5 Codespaces

I personally prefer doing Git-related things on my personal computer (offline) and then updating GitHub when I'm ready. To me, it helps avoid unnecessary merge conflicts and make sure that my edits are only committed and pushed when I'm ready.

However, you don't have to work completely on your local computer. If you don't want to have all of this software downloaded (perhaps you're running out of storage space on your computer), there is another option: codespaces. This is hosted by GitHub[63] and created an

---

[63]github.com/features/codespaces

online, virtual instance of your code files. By creating a codespace, you are using a virtual machine which runs the vs Code and is automatically connected to your GitHub repo. Codespaces could be useful if you need to edit something quickly and/or don't have access to your personal computer but make sure you create a branch before making a codespace.

If you choose to use this route, know that you are limited to 60 hours a month if you do not want to pay. You also should know that this is not available offline; an internet connection is required to work on a codespace.

So how do you make a codespace? It's easy!

1. Head to the Code tab of your GitHub repository.

2. Technically optional: create a branch. But I recommend you create a branch to lower the chance of conflicts.

3. Click on the green "Code" button (as if you were going to copy the HTTPS link for cloning).

4. In the box that pops up, click on the Codespaces tab. In this tab is a button that says "Create codespace on <branch-name>". A codespace will generate (it may take some time) and should be ready to go as soon as it's done.

B.6 Deleting a Repository

So you really to delete a repository? Think long and hard about this one. Are you sure? Are you really sure? Deleting repos are *irreversible* and can have lasting effects. For instance, all forks will be broken, but not deleted. If you delete a private repo, all forks will also be deleted. Any website that contains a link to your repo will no longer work and people might wonder where it went. And so on.

But here you are, so clearly you have decided to delete one. To delete a repository, you have to enter the danger zone (*Top Gun* anyone?).

Activity B.1 Deleting a Repository.

(a) Navigate to the repo you want to delete.

(b) Click on the Setting tab. This should be the settings for the repo itself, not the general settings for your profile.

(c) Scroll all the way down to the Danger Zone.

(d) Click on "Delete this repository"

(e) Read the dialog box that pops up. Once you have decided to delete the repo, type in what GitHub prompts you to type in and hit confirm.

Appendix C

Common Git Troubles and How To Fix Them

We all make mistakes, especially with Git. You don't know how many times I have staged the wrong files or committed my changes before realizing I did something wrong. Of coursre, it's best to try and avoid these issues becasue they can cause headaches and frustration. Plus there's usually a little fear involved (did I just erase my edits?) as well as hesitation (Git only does what I tell it, so I have to make sure I do it right). But just in case something happened that you didn't expect or you made a mistake, I have your back. Below are some common mistakes and questions you might ask.

C.1 Uh Oh, I Staged the Wrong Files!

Once you get the hang of Git, it's easy to just fly through the three-step process. In doing so, you will inevitably stage (with `git add`) a file or folder on accident. This is relatively simple to fix. To undo this, follow the next procedure.

Git Procedure C.1 Reversing `git add`. Oh shoot. What if you add a file you didn't mean to? This is one reason why there are so many steps to sending files to GitHub. Humans make mistakes and Git gives you the chance to correct any. Should you ever add a file you didn't mean to add, revist this procedure.

(a) Identify the file name(s) and extension(s) of the file(s) you mistakenly added.

(b) Use `git restore --staged <your-file-name(s)>` to un-stage any files. Like `git add`, this command can be run many times.

(c) It might be wise to verify with `git status` that everything looks good.

C.2 Uh Oh, I Committed Too Early!

We've all been there. You `add` all the files you need, double check to make sure you aren't missing anything, `commit` with a message and suddenly remember a file you should have added. Never fear, `git reset` is here!

Git Procedure C.2 Reversing `git commit`.

(a) You will only need this command: `git reset --soft HEAD^1`. It should be safe to copy/paste this command directly into your command line and run it.

(b) `git reset` gives no output so it is super important to check that the command worked with `git status` (Git Procedure 4.4, p. 51). If everything worked, you should see output as if you had used `git status` after a `git add` step.

Depending on how many commits you made, you may need to run this above command more than once. If so, make sure to use `git status` after each try!

(c) Your previously staged files should still exist. From this point, use `git add` to add your missing file(s) and proceed with `git commit`.

C.3 Git Has Been Updating Files, But Now I Don't Want It To

This is actually pretty common. Maybe you are writing a book and your abandoned chapters were on GitHub for your editors to look at. Now that the book is published, you don't want those chapters online. Maybe you are using an API and when you were developing a project had your app's API connection keys available and not that the project is public you want to remove the file containing them.

When Git is keeping track of files, we say that it is **tracking** files. Untracking files is not super difficult, but does, like all Git commands, require the command line.

Warning C.3.0.1 Untracking files from Git *does not delete them from your local computer*. However, the next time you push to GitHub, they will no longer be there. Thus, the next time your collaborators pull from the repository, their copy of the file will be deleted.

Git Procedure C.3 Untracking Files from Git.

(a) Identify the *files* you wish to remove from Git's tracking service

(b) Use `git rm --cached <filename>`. You may also add multiple files as done with `git add` (Git Procedure 4.5, p. 51).

(c) Want to untrack folders? You will need the `-r` recursive option. So, `git rm -r --cached <folder-name>` will do the trick.

(d) The above options will not delete the file(s)/folder(s) from your local computer. If you would like the files deleted and untracked, use the above commands with out the `--cached` feature: `git rm <filenames>` and `git rm -r <folder-names>`.

C.4 Git Won't Let Me Pull From Upstream or Origin!

Ah, perhaps you forgot to make a branch. This error happens to me a lot when I have a lot of work on the main branch because I never made a separate branch (not smart of me, I know).

Or maybe not. Either way, Git says you have a problem with your code and won't let you merge from upstream. Maybe you get a message like this:

```
git pull upstream main
error: Your local changes to the following files would be overwritten by
   merge:
```

```
        <file-name(s)>
Please commit your changes or stash them before you merge.
Aborting
```

This error occurs when you have local, uncommitted changes and a merge from origin or upstream has changes in the same exact places. Git has no idea whose changes to keep so it does nothing. There are a couple of different things you can do in this situation depending on how you feel about your changes.

C.4.1 Your Local Changes Are Irrelevant and You Want Them Overwritten

This definitely happens! Perhaps you tried a few things and they didn't work so you want to pull from upstream or origin and start over. Maybe you corrected some typos and someone else beat you to it so you want to merge their corrections in and keep working. I could go on and on. It's not difficult at all to force Git to make this happen, just make sure this is what you want to do!

Git Procedure C.4 Forcing an Upstream or Origin Merge.

(a) First, create a backup branch just in case things go haywire: `git branch backup`. Since `git branch` does not automatically switch you to that branch, you are free to carry on.

(b) Now, just fetch the changes. Remember that merging and fetching are separate processes. Fetching just collects the changes, attempting to merge will throw the error. Use `git fetch <remote> <branch-name>`. Example: `git fetch upstream main`.

(c) Now reset your current branch to the contents you just fetched. This is not a merge, this is a complete reset. You are reverting all of your content to match what you just fetched. This is why we created a backup in case something gets deleted that you didn't want to be.

In the event that a file you created gets deleted from the reset, just navigate to the backup branch, stage just the missing file and commit just the missing file to origin. Then pull from origin.

But to reset, use `git reset --hard <remote>/<branch-name>`. Continuing the example from above, `git reset --hard upstream/main`

C.4.2 Your Local Changes are Good and You Want to Keep Them

Sometimes your changes are valid and you want them to be kept, but you still want to pull in changes. One option is to force a merge conflict and deal with it in your editor, choosing which version to keep. If this is what you want to do, just `add` the troublesome files, `commit` them, then `pull`. Again, this will force a merge conflict but you can fix things from there.

Alternatively, you can keep your changes by using `git stash`. Stashing will store your changes locally without committing and will let you pull. To bring your changes back you can use `git stash pop`. In my experience, things have worked out quite nicely with `git stash`. I just type `git stash` before pulling, then I pull, then I use `git stash pop` and my changes are popped right back into where they were. For more information, you might find opensource.com[64] useful.

---

[64]opensource.com/article/21/4/git-stash

C.5 Git Says I Have Divergent Branches!

I recently experiences this issue for the first time and had to do some searching to figure out how to fix it. I relied on Davide Casiraghi's question on StackOverflow[65] and adrianvlupu's question also on StackOverflow[66] to create this guide and you might find these resources more helpful than mine.

This is a preemptive warning of a merge conflict and occurs when you are trying to pull in or push out a commit and another commit gets in the way. Git doesn't want to create them, so it refuses to push or pull until you fix something. This is not the same error as in Section C.4, p. 92, although it is closely related. You might get something like this:

```
git push
To <repo-you-are-pushing-to>.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to '<repo-you-are-pushing-to>.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository
    pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for
    details.
```

This looks kind of scary. Let's see how to fix this.

Git Procedure C.5 Consolidating Divergent Branches.

(a) The first step will be to try to pull in changes before you push. Use `git pull <remote> <branch>` as needed. If this works, then try to push and everything should be ok.

You might get another error like this:

---

[65]https://stackoverflow.com/questions/62653114
[66]https://stackoverflow.com/questions/29673869/

```
git pull origin main
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 659 bytes | 659.00 KiB/s, done.
From <repo-you-are-pushing-to>
   2417195..8b492ee  main        -> origin/main
hint: You have divergent branches and need to specify how to
    reconcile them.
hint: You can do so by running one of the following commands
    sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set
    a default
hint: preference for all repositories. You can also pass --rebase,
    --no-rebase,
hint: or --ff-only on the command line to override the configured
    default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

which is quite overwhelming.

(b) Here, the fix is usually pretty easy but I have to teach you a new term. We need to **rebase** the branch. When a pull happens, Git fetches new commits from GitHub and merges them into your branch (and this merge is actually a commit in itself). The rebase command will temporarily remove your new commits from the commit history, then pull in the new changes and reinstate your commit into the timeline.

If you want to avoid potential large error messages like this again, you might consider changing Git's default behavior. Instead of fetching and merging, you can do a fast-forward merge which does not do a merge commit. When a fast forward is not possible, the error message is much shorter and then you can use the rebase command to fix things.

To begin this process, first enter `git config --global pull.ff only` into your terminal. This will change the default pull behavior for Git.

(c) Now try `git pull`. Things might work. Great! But if you get an error message of `fatal: Not possible to fast-forward, aborting.`, use `git pull --rebase` to perform a rebase. Then `git push` back to origin or upstream.

If you ever want to change the default behavior back to what it was, use `git config pull.rebase false`. If you want to keep that behavior change we just did and you want to do a one-time fetch-merge pull, use `git pull --ff`.

C.6 Reverting To a Previous Commit

I have never had to do this (yet) so I don't have direct experience here. However, user gunjanpatel on GitHub[67] has created a nice guide. It appears that `git reset HEAD^ --hard` and then `git push origin -f` will work. The `-f` switch forces the push and ignores errors.

---

[67]gist.github.com/gunjanpatel/18f9e4d1eb609597c50c2118e416e6a6

Some of the guide above refers to what is known as a **commit hash**, a unique string that identifies each commit on GitHub. These are actually so unique that we only need the first few characters of the string to identify the commit. You can find the hash by using Section B.2, p. 88. The string is the random numbers/letters found on the right side of each commit.

C.7 Merge Conflicts

Hopefully you never have to deal with these. Sometimes, they are pretty simple to fix. Other times, it can cause some headaches. To avoid merge conflicts, pull from upstream often (Warning 5.4.3.1, p. 69 and Warning 6.4.2.1, p. 76). Merge conflicts are caused when more than one person edits the same line of a file. It is possible for two people to edit the same file, but on different lines, and not cause a conflict.

I have never caused or had to deal with a serious merge conflict. The GitHub documentation[68] has a good post on how to resolve conflicts on GitHub. GitHub will even show you the two (or more) versions and let you choose how to resolve them. You can also resolve merge conflicts on the command line[69] or in your text editor.

I have accidentally caused a merge conflict with myself and with working on the same repository from two different computers and two different accounts. Oops! This is an easy fix because I made the conflict with myself so I know which version is the correct one. If this happens to you, Git might send you to an unfamiliar screen when you try to push or pull. You'll know you have this case if your terminal asks you to type in a reason why the merge must happen or how the conflict is resolved. This typically happens when you try to merge from an upstream main into a local main that has active changes (so create branches!).

If you start to get stuck, do a quick Google search. There are many guides and resources out there to help with merge conflicts. Jennifer Bryan[70] has a good guide on merge conflict basics (and some other potential Git issues!) so start there and explore more as needed.

Git Procedure C.6 Merging from main to main (remote contains changes not on local). This is for when you are sent into a command line editing tool or a default text editor asking you to provide a reason why the merge is necessary.

(a) Type i

(b) Type a merge message. This can be something as simple as "I have to merge to move on" or "I know what I want, let me do it" (yes, I have actually put that as message...is this good? Who knows?).

(c) Press esc.

(d) Type :wq (this stands for write and quit)

(e) Press Enter and you should be ok to move forward.

---

[68]docs.github.com/en/pull-requests/collaborating-with-pull-requests/
addressing-merge-conflicts/resolving-a-merge-conflict-on-github
[69]docs.github.com/en/pull-requests/collaborating-with-pull-requests/
addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line
[70]https://happygitwithr.com/git-branches.html?q=merge%20confli#dealing-with-conflicts

Appendix D

Answers to All Activities and Exercises

I · Introductory Information
1 · Computer Basics
1.1 · File and Folder Names
1.1.2 · Exploring File and Folder Naming Techniques

Activity 1.1 Folder Naming.

(a) **Solution**.   The main thing that pops out to me are the spaces. Every multi-word folder name has a space. Secondly, some of the names are very long. Third, there is not much consistency. Some names are capitalized, others aren't. There are clearly deluxe versions of albums but some say "deluxe", others say "deluxe version", and even others say "deluxe edition".

The three here are the main issues; you may have thought of others. Great! I'm sure they are good observations.

(b) **Solution**.   There are probably a lot of patterns you could have noticed. I grouped folder names into

    1 Standard albums

    2 Deluxe albums

    3 Live albums

    4 Taylor's Version albums

    5 Other

(c) **Solution**.   Since there aren't any special modifiers to this album (such as deluxe or live), I say we just keep its name, but write them in kebab-case. Therefore, "Taylor Swift" would become `taylor-swift`.

(d) **Solution**.   The following names are the modified versions:

- `taylor-swift`
- `fearless`
- `speak-now`
- `red`
- `1989`
- `reputation`
- `lover`
- `folklore`
- `evermore`
- `midnights`

(e) **Solution**. For the deluxe albums, the similarity is the word "deluxe". I don't bother with "edition" or "version", "deluxe" is enough to convey the necessary information. I also would like to abbreviate "deluxe" to "dlx". You could also choose to keep the full word, but since we can't ask Taylor what she would prefer, I am sticking with the shorter. "dlx" gets the point across (this is a deluxe album) with 3 less characters. Following are my changes for the deluxe group. There is one outlier: the Fearless Platinum Edition. I am going to retain the "platinum" name since that is its true name.

- `taylor-swift-dlx`
- `fearless-pltnm`
- `speak-now-dlx`
- `red-dlx`
- `1989-dlx`
- `folklore-dlx`
- `folklore-dsny-dlx`
- `evermore-dlx`

(f) **Solution**. There are only two in this group, but there could be more in the future. Let's use "live" to differentiate these folders from the others. I am choosing to not use "lve" or "lv" because I think there are slightly ambiguous (it could mean "love"!).

- `clear-chnl-live`
- `speak-now-live`

(g) **Solution**. I am going to again use kebab-case and use "tv" to indicate that the album is a Taylor's Version. This will help cut down on name length.

- `fearless-tv`
- `red-tv`

(h) **Solution**. These are mostly up to you. They have no specific pattern, but I will still apply kebab-case and shorten the names.

- `holiday-clctn` (I could even use `holiday` since there is only one holiday album)
- `unreleased`
- `singles`

Activity 1.2 File Naming.

(a) **Solution**. The main thing that pops out to me are the spaces. Every multi-word file name has a space. Secondly, there are a few cases of non-consistency: Back To December says "Acoustic" whereas Haunted says "Acoustic Version".

The two here are the main issues; you may have thought of others. Great! I'm sure they are good observations.

(b) **Solution**. There are probably a couple patterns you could have noticed. I grouped track names into

1 "Regular" album tracks
2 Acoustic tracks
3 The last track

(c) **Solution**. There isn't much we can do here. I will write the names in snake_case which will remove the spaces and change every word to lowercase. Since we can't ask Taylor for her preferences, it is up to you whether you want to abbreviate words or not. For the most part, I do not since most track names are short already.

- 01_mine.wav
- 02_sparks_fly.wav
- 03_back_to_dec.wav
- 04_speak_now.wav
- 05_dear_john.wav
- 06_mean.wav
- 07_story_of_us.wav
- 08_never_grow_up.wav
- 09_enchanted.wav

- 10_better_than_revenge.wav
- 11_innocent.wav
- 12_haunted.wav
- 13_last_kiss.wav
- 14_long_live.wav
- 15_ours.wav
- 16_if_this_was_movie.wav
- 17_superman.wav

(d) **Solution**. I am choosing to abbreviate "acoustic" to "acoust". Notice I used the same names as the regular tracks, but just appended "_acoust" at the end.

- 18_back_to_dec_acoust.wav
- 19_haunted_acoust.wav

(e) **Solution**. I am choosing to abbreviate "POP Mix" to "pop_mix". Notice I used the same name as the regular tracks, but just appended "_pop_mix" at the end.

- 20_mine_pop_mix.wav

2 · The Command Line
2.3 · Basic Commands
2.3.1 · Navigating Your Computer With the Command Line

Checkpoint 2.3.1.3 Trying `cd`.

Hint. Try running `pwd` after `cd`. Do you notice anything different here?

2.3.2 · Adding Files and Directories

Activity 2.1 File Extensions Matter.

(c) **Answer**. Mac users likely see the TextEdit app open. This is the default app for opening files like this. Windows users will be prompted to determine which application should be used to open the file.

Checkpoint 2.3.2.2 Trying `open`.

Answer. `open test.txt` or `start text.txt`

Checkpoint 2.3.2.3 Trying `killall`.

Solution. Your computer should have opened up TextEdit (the default text editor for Mac). To close this application, you should enter `killall TextEdit` at the command line.

Activity 2.2 Putting It All Together, Part 1.

(a) **Answer**. `cd testdir` should do the trick!

(b) **Answer**. Use `ls` and `pwd` here.

Solution. Since we just created this directory, `ls` should not show any files; no output should be provided. `pwd` can be used to find the file path which should look like

```
/Users/<your-username>/Desktop/testdir
```

or

```
/c/Users/<your-username>/Desktop/testdir
```

(c) **Hint**.   You can use `ls` to verify that everything worked.

Answer.   `touch my_greeting.txt` OR `touch myGreeting.txt` OR `touch my-greeting.txt` OR `MyGreeting.txt` OR `MY_GREETING.txt`

Solution.   Whenever we create a new file (of any type), we use `touch`. Recall that `touch` requires three pieces: the command, the file name, and the file extension. The command is `touch`, the file name is `my_greeting` (recommended, although there are other possible names), and the file extension is `.txt`. Put these together, and we get `touch my_greeting.txt`.

(d) **Answer**.   `open my_greeting.txt` OR `open <file-name>.txt`

(e) **Answer**.   This is a `.txt` file: `killall TextEdit`.

2.3.3 · Removing Files and Directories

Checkpoint 2.3.3.1 Trying `rm`.

(b) **Answer**.   `rm my_greeting.txt`

Checkpoint 2.3.3.3 Trying `cd` Backwards.

Hint.   If successful, `pwd` should indicate that you are in the Desktop.

Answer.   `cd ..`

Solution.   As mentioned, `cd ..` will move you backwards. If you run `pwd`, you should see that you have returned to the Desktop; `ls` should give you all of the files we were working with before.

Checkpoint 2.3.3.4 Trying `rmdir`.

Answer.   `rmdir testdir`

Activity 2.3 Putting It All Together, Part 2.

(a) **Answer**.   Use `pwd` to check if still on Desktop, `rm <file-name>` to remove files, and `ls` to check if the files are gone.

(b) **Answer**.   `cd ..`

(c) **Answer**.   `cd Documents` OR `mkdir Documents`, then `cd Documents`

(d) **Answer**.   `mkdir my-favorites` (could use a different case)

(e) **Answer**.   `cd my_favorites`, `pwd`

(f) **Answer**.   `touch food.txt`, `touch hobbies.docx`, `touch smells.xslx`; `ls`

(g) **Answer**.   Mac: `open food.txt`, `killall TextEdit`; `open hobbies.docx`, `killall 'Microsoft Word'`, `open smells.xslx`, `killall 'Microsoft Excel'`

Windows: `start food.txt`, `start hobbies.docx`, `start smells.xslx`

(h) **Answer**.   `rm food.txt`, `rm hobbies.docx`, `rm smells.xslx`; `ls`

(i) **Answer**.   `cd ..`, `pwd`

(j) **Answer**.   `rmdir my-favorites`, `ls`

II · Working Solo
4 · Git Solo

4.3 · Sending Changes Back To GitHub: The Three-Step Process
4.3.1 · Wait, Which Files Did I Change Again?

Git Procedure 4.4 Trying `git status`.

Solution.   You might see a console output such as the one below.

```
git status
On branch country
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We get a lot more information than needed right now, but we will break this output down shortly. For now, notice that we get a statement of the branch we are on and a file name that represents a file we have modified. It is not recreated here, but the modified file name is red in the output (any new, untracked files will be green).

4.3.2 · Step 1: Staging Files

Git Procedure 4.5 Trying `git add`.

(b) **Hint**.   Case 1 or Case 3 will work in this instance.

   Answer. `git add README.md` OR `git add .`. Neither will produce any output.

(c) **Answer**.

```
git status
On branch country
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
```

The text color has changed to green which indicates the process worked.

4.3.3 · Step 2: Committing Files

Git Procedure 4.6 Trying `git commit`.

(c) **Answer**.

```
git status
On branch country
nothing to commit, working tree clean
```

This output is what is to be expected! Don't fret that it says nothing to commit. This is completely true since we just finished committing everything that we had staged. This output means that everything worked correctly.

4.3.4 · Step 3: Pushing Files

Git Procedure 4.7 Trying `git push`.

(a) **(i) Solution**.

```
git push
fatal: The current branch country has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin country
```

Well that looks a little scary. Fatal! That just means that git wansn't able to complete your request and stopped trying. The reason this failed is because the branch country is only on your computer, it hasn't yet made it to GitHub (which is what we are trying to do). It is possible to create a remote connection between branches but I often don't especially if I am only working on one branch at a time and if I plan on deleting a branch when I'm done with it. Should you want to create a remote, use the command Git provided.

(ii) **Solution**.

```
git push origin
fatal: The current branch country has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin country
```

Well, look at that, the same error. This should make sense. You are telling Git to push changes to the origin repo but country, the current branch, doesn't have an origin since we created the branch locally.

(iii) **Solution**.

```
git push country
fatal: 'country' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Remember the four parts to git push? Well since we forgot Step 3, the remote name, Git assumes that the word "country" is the name of the repo we want to push to. Since there is no repo named "country" on your computer nor on GitHub, it doens't know where to push and errors out.

(b) (i) **Answer**. Here, to follow with the example, we will use origin and the branch name country.

(ii) **Solution**.

```
git push origin country
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 4 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 330 bytes | 82.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local
    object.
remote:
remote: Create a pull request for 'country' on GitHub by
    visiting:
remote:      <url-to-pull-request>
remote:
To <url-to-origin-repo>
 * [new branch]      country -> country
```

We get a lot of output for a successful push. Most of it is just information that can be useful in certain cases and might be worth a glance to make sure things worked as expected.

4.4 · The Final Steps

Git Procedure 4.10 Trying `git pull`.

(b) Answer. `git switch main`

(c) Answer.

```
git pull origin main
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 631 bytes | 631.00 KiB/s, done.
From <repo-url>
   <commit-id>  main        -> origin/main
Updating <commit-id>
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
```

4.6 · Working Solo: The Culminating Experience

Activity 4.11

(a) Answer. Use one of the multiple methods to create a new repo in GitHub. Perhaps name it `poems` or `fav-poems`? This part is up to you as long as the name accurately describes the future repo contents. Check the box that says to add a README file.

(b) Hint. Make sure you are not "inside" your previous repositories. Nested repositories will not work. So `cd` to the folder in which you want to place your new repo.

Answer. Copy the HTTPS link on GitHub. Use `git clone <copied-link>`. Use `cd <repo-name>` to move your terminal into that repo.

(c) Answer. `git branch readme` then `git switch readme` OR `git switch -c readme`

You may use any branch name you like as long as it's clear what the branch is for. "readme" seems like it will work nicely.

(d) Answer. Perhaps a better title would be "My Favorite Poems" or "Two Original Poems". This can vary. You might explain that this repository contains two of your favorite poems (or two poems you wrote yourself) and this it was born from a project in *Better Git Started* to help you get more familiar with GitHub and Git functions. This can also vary.

(e) Answer. `git status`

(f) Answer. `git add .` OR `git add README.md`, maybe `git status` to confirm?

(g) Answer. `git commit -m 'edit readme with repo info'`, maybe `git status` to confirm?

Of course, your commit message may be different. Perhaps "edit readme" and enough for you.

(h) Answer. `git push origin <your-branch-name>`

(i) Answer. "Compare and pull request", "Merge", "Confirm merge". (Check the main branch on GitHub to make sure this worked.)

(j) **Answer**. `git switch main` then `git pull origin main`

(k) **Answer**. `git branch poems` then `git switch poems` OR `git switch -c poems`

You may use any branch name you like as long as it's clear what the branch is for. "poems" works for me.

Note that you could split up each poem into its own branch. Depending on the length of your poems, you may want to do that in practice but here we will edit both on one branch. That way you can practice two commits in one push!

(l) **Hint**. In vs Code, you can create a new file with the paper-with-a-plus icon in the "Explorer" panel in the top left.

Answer. Use the hint or use your terminal (`touch poem1.md`).

(n) **Answer**. `git add .` OR `git add poem1.md`, then `git status`

Your text editor might also indicate that this worked. For instance, vs Code will show an "A" next to the file name.

(o) **Answer**. `git commit -m "<commit-message>"`, `git status`

Your text editor might also indicate that this worked. For instance, vs Code will remove and letters and colors next to the file name. It will also show your commit message in the "Timeline" panel on the bottom left.

(p) **Answer**. After editing, `git add .` or `git add poem2.md`. Then `git commit -m "<message>"`. `git status` might be useful here.

(q) **Answer**. `git push origin <branch-name>`, "Compare and pull request", "Merge", "Confirm merge".

(r) **Answer**. `git switch main`, then `git pull origin main`

III · Working With Others
6 · Git Collaboration
6.1 · Getting Set Up: A Few Extra Steps
6.1.1 · Cloning A Fork

Checkpoint 6.1.1.1 Solution. Use the green code tab *on your fork* to copy the HTTPS link. Then use `git clone <https-link>`. Make sure you have used `cd` to get to the right place first!

6.1.2 · Don't Forget About Branches!

Checkpoint 6.1.2.1 Solution. There are a couple of choices here.

1. `git branch <name>` then `git switch <name>`

2. `git switch -c <name>`

6.4 · The Final Steps
6.4.2 · Pulling/Fetching Upstream

Git Procedure 6.1 Updating Origin and Local, Method 1.

(c) **Answer**. `git pull origin main` (if you are pulling from origin, just `git pull` can work, too)

6.6 · Working Together: The Culminating Experience

Activity 6.3

(b) Answer. Click on the "Fork" button in the top right. (You can keep the same name.)

(d) Answer. Copy the HTTPS link *to your fork* on GitHub. Use `git clone <copied-link>`. Use `cd <repo-name>` to move your terminal into that repo.

(e) Answer. This could vary. Perhaps `superpower`, `power`, or `add-superpower`?

(f) Answer. `cd book-activities` first, then `code superpower.md` to open just the markdown file (not the entire repository) (if using VS Code); if using a different editor, use Google to find a good terminal command (or do it manually)

OR `cd ..` to go backwards a folder, then `code gitstarted` to open the entire repository, then open the markdown file using work (to use Git functions again, you may have to `cd gitstarted` again)

(h) Answer. `git status`

(i) Answer.

- `git add .` or
- `git add book-activities/` or
- `git add book-activities/superpower.md`

(j) Answer. `git status`

(k) Answer. `git commit -m '<your-message>'`

(l) Answer. `git push origin superpower`

(m) Answer. Create a pull reqeust from `superpower` to your fork's `main` then from your fork's `main` to upstream `main`.

OR Create a pull request from `superpower` directly to the upstream `main`.

(o) Answer. `git switch main`, then

`git remote add upstream https://github.com/ian-curtis/gitstarted.git`

It is important to switch back to main or you will set an upstream remote for just your branch which is sometimes useful, but not here.

(p) Answer. (on main branch) `git pull upstream main`, `git push origin main`

Index

Colophon

This book was authored in PreTeXt.