

# Git and GitHub for Public Health

Corinne Riddell and Lauren Wilner

2024-06-17



# Contents

<b>1 Git and GitHub for Public Health</b>	<b>5</b>
1.1 Overview . . . . .	5
1.2 Contributors and Acknowledgements . . . . .	5
<b>2 Workshop Setup</b>	<b>7</b>
2.1 Introduction . . . . .	7
2.2 Install R and RStudio . . . . .	7
2.3 Open RStudio and install the following packages . . . . .	8
2.4 Create a GitHub account . . . . .	8
2.5 Install Git . . . . .	8
2.6 Configure Git using an HTTPS token . . . . .	9
2.7 Download data and code used for the workshop . . . . .	10
2.8 Resources . . . . .	10
<b>3 Why Git and GitHub</b>	<b>11</b>
3.1 What is version control, Git, and Github? . . . . .	11
3.2 The case for version control . . . . .	11
<b>4 Git Jargon</b>	<b>13</b>
<b>5 Interacting with Terminal and Bash</b>	<b>15</b>
5.1 Terminal primer video . . . . .	15
5.2 A Terminal window . . . . .	15
<b>6 Naming folders and files</b>	<b>19</b>
<b>7 Solo Workflow</b>	<b>23</b>
7.1 Outline for this section . . . . .	23
7.2 Set up a project that you want to track with Git and GitHub . .	23
7.3 Make your first branch . . . . .	25
7.4 Moving forward: More practice with branching . . . . .	28
7.5 Another practice: Changes after receiving an updated dataset .	32
7.6 Summary . . . . .	36

<b>8 Group Workflow</b>	<b>37</b>
8.1 Outline for this section . . . . .	37
8.2 Setup . . . . .	37
8.3 Get oriented to the new directory . . . . .	38
8.4 Collaborative Routes . . . . .	38
<b>9 What to track on Git and Github?</b>	<b>47</b>
9.1 Approach 1 : Track statistical code only . . . . .	47
9.2 Approach 2: Track statistical code, some data files, images, and other documents . . . . .	47
9.3 Things you definitely do not want to track . . . . .	50
<b>10 Daily Git Practice</b>	<b>51</b>
<b>11 Assorted Git processes and tips</b>	<b>53</b>
11.1 Forking . . . . .	53
11.2 Ignoring files . . . . .	55
11.3 Rolling back changes . . . . .	56
11.4 And if all else fails... . . . . .	56

# Chapter 1

# Git and GitHub for Public Health

This is the website for the training “**Git and GitHub for Public Health**”, first offered in 2024.

## 1.1 Overview

Keeping track of changes to your statistical code is key to reducing errors and boosting teamwork in your projects. However, many public health professionals, including epidemiologists, haven’t been trained on how to manage these changes, which might leave you wondering about the integration with institutional review board (IRB) protocols and privacy rules. On this website, we dive into the basics of Git and GitHub. We’re here to show you how these tools can help you confidently and ethically manage your code changes, making your work and research smoother and more secure.

## 1.2 Contributors and Acknowledgements

Corinne Riddell and Lauren Wilner created this website. Contact us at [c.riddell@berkeley.edu](mailto:c.riddell@berkeley.edu) and [wilnerl@uw.edu](mailto:wilnerl@uw.edu) if you have questions, comments, or feedback.

This website references and builds upon the work of Jenny Bryan, especially the Happy Git and GitHub for the useR website. This is an invaluable reference and one that we referred to numerous times when creating this website.

This website is hosted by <https://www.netlify.com>.



# Chapter 2

## Workshop Setup

This pre-workshop guide is designed to walk you through the initial setup of Git on your computer.

If you have any issues, please reach out to us at [c.riddell@berkeley.edu](mailto:c.riddell@berkeley.edu) and [wilnerl@uw.edu](mailto:wilnerl@uw.edu).

### 2.1 Introduction

Git is a version control system that allows you to track changes in your code.

In order to get setup, you need to install Git on your computer, make a GitHub account, and then configure Git on your computer.

If you are attending a live workshop, we will cover the details of how to use Git during the workshop. To ensure the workshop runs smoothly and efficiently, we request you to complete the following steps before the workshop – this enables us to spend all of the workshop time on using Git rather than on setup. Thank you!

### 2.2 Install R and RStudio

If you don't already have R and RStudio please install them. While the workshop will also give options using SAS, everyone needs R and RStudio for some pieces of the workshop – it should only take a few minutes to download and install each of these.

**Link to both downloads:** <https://posit.co/download/rstudio-desktop/>

If you download each of these a while ago, we recommend that you update the versions so everything runs smoothly.

## 2.3 Open RStudio and install the following packages

- tidyverse
- usethis
- gitcreds
- broom

To do this, run the following code in the RStudio console (lower left-hand side panel, of the four panel pane):

```
install.packages('tidyverse')
install.packages('usethis')
install.packages('gitcreds')
install.packages('broom')
```

*If any of these packages fail to install, please let us know.*

If this is your first time using R/RStudio, this video shows how to type the commands into the console and R looks like when the installs runs smoothly. *Please view this video in full screen and at high quality to see it properly.*

## 2.4 Create a GitHub account

<https://github.com/>

## 2.5 Install Git

Installing Git is different depending on whether you are using a Windows or Mac. Follow the relevant instructions.

### 2.5.1 Windows Instructions

- 1) Download Git for Windows: <https://git-scm.com/download/win>. Once the download begins, there are many menu items to click through. As recommended on the fantastic website HappyGitWithR, accept the default selections except for the item “Adjusting your PATH environment”. There, make sure to select “Git from the command line and also from 3rd-party software”. HappyGitWithR also notes that “RStudio for Windows prefers for Git to be installed below C:/Program Files and this appears to be the default. This implies, for example, that the Git executable on my Windows system is found at C:/Program Files/Git/bin/git.exe. Unless you have specific reasons to otherwise, follow this convention.

The following video shows how to download and install Git for Windows. View it on full screen and in high quality if you want to follow along:

- 2) Once you have Git downloads, confirm you have access to Git Bash. To do so, it should appear in your Start menu: Start menu > Git > Git Bash.
- 3) RStudio should automatically detect the presence of Git Bash. You can inspect and influence this directly via Tools > Global Options > Terminal. Unless you have good reason to do otherwise, you want to see “Git Bash” in the “New terminals open with ...” dropdown menu.
- 4) The next set of tasks are done in RStudio and outlined below in the section “Configure Git using an HTTPS token” below the Mac instructions.

### 2.5.2 Mac Instructions

- 1) Open up the Terminal application, and type `git --version` to check if Git is already installed. If it is, you will see a version number. If it is not, type: `git config` and then press return. You will be prompted to install Git and follow the prompts!

## 2.6 Configure Git using an HTTPS token

In RStudio, load the libraries `usethis` & `gitcreds`:

```
library(usethis)
library(gitcreds)
```

Use the function `use_git_config()` to configure Git using an HTTPS token. To do so, type the following in the RStudio Console (lower left panel), filling in a user name, and email. The user name is up to you – it doesn’t need to be your GitHub user name. `user.email` is the email address linked to your GitHub account.

```
use_git_config(user.name = "Your user name",
               user.email = "your.email.adress@gmail.com")
```

Next up, you will need to create a personal access token. This is a way to authenticate yourself with GitHub. You will need to do this in order to push and pull from your GitHub repository. To do so, run the following in the R console:

```
usethis::create_github_token()
```

This will bring you to a browser page. Put in a description for your token and then select an expiration date from the drop down - please select `No expiration`. Scroll down and click the `Generate token` button. Copy the token that is generated and paste it somewhere where you will be able to access it.

Go back to R/RStudio and run the following:

```
gitcreds::gitcreds_set()
```

When prompted, paste in the GitHub token you copied from the website. This will add your credentials to your cache. The following will print out to the RStudio console:

```
? Enter password or token: ghp_xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
-> Adding new credentials...
-> Removing credentials from cache...
-> Done.
```

## 2.7 Download data and code used for the workshop

Run the following two commands in the RStudio console:

```
install.packages("usethis")
usethis::use_course("corinne-riddell/existing-project")
```

- R will ask you if you want this folder copied onto the Desktop. Select Yes.
- R will display messages showing you that the folder has been downloaded and unzipped. Tell R whether to delete the file.
- RStudio will launch.
- If you are attending a live workshop, keep this folder handy – we will reference it then. If you are following along asynchronously we will use this folder in the solo workflow section.
- If you would like, feel free to run the code in the folder, but do not otherwise edit it at this time.

## 2.8 Resources

You are now set up to use Git and GitHub! If you had any issues, here are a few links you can look at for help. If you are still having trouble, please reach out to us before the workshop.

- Happy Git with R
- Git Setup Book

If you are attending the live workshop, we will cover the rest of this website live. See you there!

# Chapter 3

## Why Git and GitHub

### 3.1 What is version control, Git, and Github?

**Version control** is the practice of tracking and managing changes to (statistical) code and other files.

**Git** is a version control system. It tracks what is changed in a file, when and by whom and synchronizes the changes to a central server so that multiple contributors can manage changes to the same set of files (Wilson et al., 2017).

**GitHub** is a hosting service on the web for Git repositories.

### 3.2 The case for version control

#### Reduces Errors

Version control **eliminates** the need to send code or outputs (graphics, reports) via email or share folders between collaborators. With a version control system, everyone has access to the most recent set of files.

#### Facilitates reproducible analyses

Have you ever tried to reproduce an analysis you did 3 years ago? Have you ever tried to reproduce someone else's analysis? Have you ever found their links don't work on your computer? Or that you do not have access to the data?

- Because everyone has access to the same files, a project's workflow can be set up to ensure that the analyses are reproducible for everyone.
- In R, this can be as simple as hitting the "knit" button to run the analyses on anyone's computer – no need to update the file pathways, no need to download new versions of the code or data.

#### Eases and supports supervision and collaboration

With Github, you can easily view changes made to statistical code. So if you are working together, it is easy to tell what lines of code were changed, alongside downstream changes to reports or data visualizations as a result of the change to the analysis.

**Facilitates rollbacks**

You can use Git to roll back to a previous version of a file at any point. This is useful if your team decides that a change made to the code was not beneficial or wants to revert back to a different strategy that was used previously.

**Supports expeditious sharing of scientific approaches and findings**

Anything posted on GitHub can be shared widely to your organization or with the public.

# Chapter 4

## Git Jargon

Before diving into using Git and GitHub, we need to become immersed in its jargon. Fortunately, Alice Bartlett, a Principal Engineer at the Financial Times, made an excellent set of slides on Git and GitHub and the associated jargon. She covers:

- repository, or “repo”
- commit
- hash
- checkout
- branch
- merge
- remote
- clone
- push
- pull

In the live workshop, we will go through Alice’s slides together. If you are going through this training asynchronously, you can flip through Alice’s slides here:

[Download slides here](#)



# Chapter 5

## Interacting with Terminal and Bash

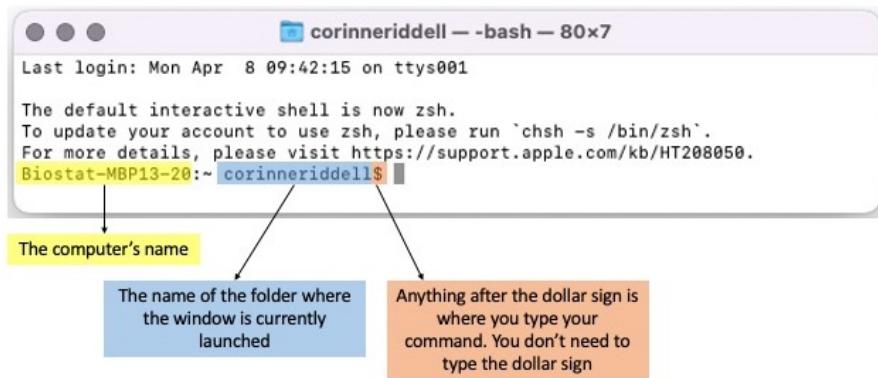
We have two more things to cover before we dive in to Git and Github. First, we will be using Mac's Terminal and Window's Bash for this training. If you have never used these applications before, we introduce them below, so you become familiar with them. Second, we discuss good names for files and folders, as good names make for good times when you interact with Terminal and Bash.

### 5.1 Terminal primer video

If you have never used Mac's Terminal or Window's Bash applications before, please watch this 9-minute video on how to use these applications to navigate to your project folders. While this was recorded on Mac, the same concepts apply for Windows.

### 5.2 A Terminal window

We will use Terminal and Bash applications to interact with Git on our laptops. Below is a Mac Terminal window. It looks very similar to a Windows Bash window.



In this training, we supply you with Git code for you to type into your Terminal/Bash windows.

Once you input a command and hit Enter/return on your keyboard, Terminal/Bash will display output to the screen. Pay attention to this output as it will indicate if the command was understood by your computer or if something went wrong. For example, here is the command `git branch`, followed by the output printed to screen:



Here is a cheat sheet of commands we use on this website:

Mac Command	Windows Command	Description
<code>ls</code>	<code>dir</code>	Lists files and directories
<code>cd {dir}</code>	<code>cd {dir}</code>	Changes to directory {dir}
<code>cd ..</code>	<code>cd ..</code>	Moves up one directory level
<code>open .</code>	<code>start .</code>	Opens the current directory
<code>pwd</code>	<code>pwd</code>	Prints the current working directory

Here is a cheat sheet of commands we don't use but are helpful to know and integrate into your workflow:

Mac Command	Windows Command	Description
git rm {file}	git rm {file}	Removes a file
mkdir {}	mkdir {}	Creates a new directory {}
touch {}	type {file}	Creates a new empty file {}
cp {source} {destination}	copy {source} {destination}	Copies files or directories from {source} to {destination}
git mv {source} {destination}	git mv {source} {destination}	Moves or renames files or directories from {source} to {destination}
history	history	Shows the command history
clear	cls	Clears the terminal screen



# Chapter 6

## Naming folders and files

When you are working in Terminal/Bash, it is important to use good folder and file names to make your life easier.

### 1. Good names for folders: no spaces or special characters

Recall when you name a variable in SAS or R, the variable name cannot contain spaces or unusual characters. **It is best practice to not use spaces or unusual characters in folder or file names**, even though spaces are permissible and commonly used by Windows and Mac Users.

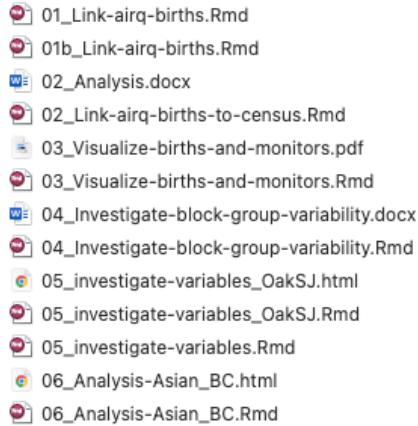
You may wonder what the problem is with spaces, anyway? While spaces are human-readable they aren't machine-friendly. When you refer to a folder or file using Git in Terminal or Bash, a name without spaces is much easier to type (otherwise you have to insert a backslash before the space). Spaces also break the auto-complete function that Git users love. This is frustrating.

Good folder names and files names **use dashes and underscores in place of spaces**. For example “life-expectancy” is a great folder name for a project estimating life expectancy, while “pollution-ptb” is a great folder name for a project estimating the causal effect of air pollution on preterm birth.

### 2. Good names for code files: numbered with underscores and dashes

When running a statistical analysis, there is an underlying order across the code files (e.g., starting with importing the data, then data cleaning, and analysis). Good code file names should start with a number (padded by 0) to order the files according to the order they are performed in the analysis. This is followed by a short (human and machine readable) descriptor of what the file does. You can also use underscore “\_” to delimit field, and dashes “-” to separate words within field.

For example, here are some of the file names in the “pollution-ptb” folder:



These files are:

- machine readable
- human readable
- play well with default ordering

Flip through Jenny Bryan’s slide set here for a deeper dive into file naming:

[Download slides here](#)

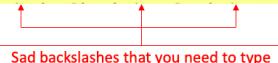
### 3. What happens when there are bad names?

The issue and frustration of includes spaces in file names really hits home when you include spaces in the names and trying interacting with GitHub to add the files and make a commit – if you like pain and frustration – you can try this at home!

For example, suppose I made an R markdown file called “Data Visualization Evaluation Report.rmd”. In the screenshot below, you can see that this file has been modified when I used the `git status` command. The issue occurs when I need to add that file using `git add {file_name}`. If I typed `git add Data Visualization Evaluation Report.rmd` there would be an error because Git doesn’t know that the name continues after the space after the word “Data”! Even worse, the space breaks the auto-complete functionality that occurs when you press “tab” to auto-complete the file name after typing the first few letters.

To really add the file we need to use the escape character (the backslash “\”) before each space in the name, as shown in the `git add` command highlighted in yellow. This might not sound like a big deal if you haven’t interacted with Terminal/Bash very much, but I can guarantee you that the frustration will build to a crescendo over time... save yourself this pain by using dashes and underscores instead of spaces!

```
| Biostat-MBP14-02:course-evaluations corinngeriddell$ git status  
On branch cr-apr8  
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: reports/Data Visualization Evaluation Report.Rmd  
    modified: reports/Data-Visualization-Evaluation-Report.pdf  
  
Untracked files:  
  (use "git add <file>..." to include in what will be committed)  
    .DS_Store  
    .gitignore  
    code/ReadlinkedData.html  
    data/.DS_Store  
  
no changes added to commit (use "git add" and/or "git commit -a")  
| Biostat-MBP14-02:course-evaluations corinngeriddell$ git add reports/Data\ Visualization\ Evaluation\ Report.Rmd
```



Culprit file name with spaces

Sad backslashes that you need to type before every space in a file name ☺



# Chapter 7

## Solo Workflow

### 7.1 Outline for this section

We have discussed why Git and GitHub are important, now we will set up a repository and work through an example. During this first section, you will be working in a repository alone. We will:

- Make new repository on GitHub
- Clone the repository to your local machine
- Write code in your repository locally
- Push the code to your repository
- Merge your branch into main
- Repeat the process with more code updates
- Repeat the process with data updates

### 7.2 Set up a project that you want to track with Git and GitHub

Let's suppose that you have a project that you want to start tracking using Git and Github. For this project, you are already working on with some code, data, and visualizations that have already been saved. We have made this project for you. It is the one called "existing-project" that should be on your Desktop. If it is not, go back to the Workshop Setup page and run the `usethis::use_course("corinne-riddell/existing-project")` code to begin this section.

The `existing-project` folder contains existing code and things that you might want to start tracking on GitHub. The next thing to do is make a folder on GitHub to store this project.

### Make a new repository on GitHub

- Go to [github.com](https://github.com) and log in. Click the green “New” button to make a GitHub repository. Type “life-expectancy” in the repository name.
- Write whatever you want in the description. For example, type “An analysis of life expectancy in the US”
- Select either to make this a public or private repository.
- Check the box next to **Add a README file**. This tells Git to create a file that can describe your project. For now, you can write a sentence about this being a practice repository for this workshop.
- Choose `.gitignore` template: R. This tells Git to use defaults that work well for R users.
- Keep the license as None. (You would want a license if you are developing code for software, like an R package, but not for every day analysis code).
- Click “Create repository”. Github will then bring you to the repository’s main page.

### Clone the repository to your local machine

- From the main page of your repository and click on the green **Code** button.
- You’ll see a URL that starts with `https://`. Push the icon with two overlapping squares to copy the URL to your clipboard.
- Open Terminal (Mac) or Bash (Windows) program. Navigate to where you want to place this repository using the `cd {folder_name}` command.  
*Note: Whenever we use curly brackets on this website, you do not need to type out curly brackets in your code. For example use `cd Documents/` to navigate to your Documents folder, not `cd {Documents}`. This applies wherever you see curly brackets on this website!*
- Then type `git clone {paste the url you copied here}` in Terminal/Bash and then press the return/enter button. The following will display in Terminal/Bash if this was successful:

```
Biostat-MBP13-20:repos corinneriddell$ git clone https://github.com/corinne-riddell/life-expectancy
Cloning into 'life-expectancy'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), done.
```

You now are ready to begin tracking changes to this folder using Git and GitHub.

### Get oriented to the new directory

To get yourself oriented, do the following in the Terminal/Bash window:

- Navigate into your repository by typing `cd life-expectancy/`.
- Type `git status`. The results shows you that no changes have been made yet:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch main
Your branch is up to date with 'origin/main'.
```

`nothing to commit, working tree clean`

- Type `git branch`. This shows you that you are currently on the main branch.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
* main
```

## 7.3 Make your first branch

Set yourself up in a new branch off of main. In Terminal/Bash:

- Type `git checkout -b june18-xy`, replacing xy with your initials – write these in lowercase to prevent errors later! (If today is not June 18th, replace “june18” with today’s date.)

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git checkout -b june18-cr
Switched to a new branch 'june18-cr'
```

- Type `git branch`, to confirm to yourself that you have indeed switched to the new branch.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
  main
* june18-cr
```

### Make some changes to your tracked folder

Okay, you are now set up to track changes. Let’s do the following:

Copy the code/, data/, and images/ sub-folders from your “existing-project” folder into the “life-expectancy” folder as well as the .Rproj file.

### Commit the changes that you made and push them to GitHub

Go back over to Terminal or Bash. Type `git status`. The output will tell you what has been changed. It tells us that there are untracked files:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june18-cr
```

```
Untracked files:
(use "git add <file>..." to include in what will be committed)
.DS_Store
code/
data/
images/
existing-project.Rproj
```

We want to track the code/, data/, and images/ subfolders we just copied over, as well as the existing-project.Rproj file.

Use `git add` to add the newly-added files to be tracked. Then use `git status` to confirm you have added everything you want to track:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git add code/
Biostat-MBP13-20:life-expectancy corinneriddell$ git add data/
Biostat-MBP13-20:life-expectancy corinneriddell$ git add images/
Biostat-MBP13-20:life-expectancy corinneriddell$ git add existing-project.Rproj
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june18-cr
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
 new file:   code/01_Analyze-life-expectancy.R
 new file:   data/Life-expectancy-by-state-long.csv
 new file:   images/ca-black-women-LE.png
 new file:   images/placeholder.md
 new file:   existing-project.Rproj
```

```
Untracked files:
(use "git add <file>..." to include in what will be committed)
.DS_Store
```

Note: Computers create files that we don't want to track. For example, Macs create .DS\_Store files. Another example is that Windows creates temporary files when a Word doc or Excel spreadsheet is open. You will see these weird files listed under the Untracked files list. You don't need to worry about them because we don't want to track changes to any of those files.

Commit these changes locally: `git commit -m 'your commit message'`, replace 'your commit message' with a short message about what you've done (keep the quotes around the message). For example, your message could be something like `git commit -m "added first set of files"`.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git commit -m "added first set of files"
[june18-cr 58fcc58] added first set of files
5 files changed, 7253 insertions(+)
create mode 100644 code/01_Analyze-life-expectancy.R
create mode 100644 data/Life-expectancy-by-state-long.csv
create mode 100644 images/ca-black-women-LE.png
```

```
create mode 100644 images/placeholder.md
create mode 100644 existing-project.Rproj
```

Push these changes to GitHub: `git push origin {branch-name}`, replacing `{branch-name}` with the name of your branch. If you don't remember your branch's name, type `git branch` to print it to the screen and then the `git push` command.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git push origin june18-cr
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (9/9), done.
Writing objects: 100% (10/10), 136.50 KiB | 10.50 MiB/s, done.
Total 10 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'june18-cr' on GitHub by visiting:
remote:     https://github.com/corinne-riddell/life-expectancy/pull/new/june18-cr
remote:
To https://github.com/corinne-riddell/life-expectancy.git
 * [new branch]      june18-cr -> june18-cr
```

You have successfully pushed your changes to GitHub!

### Merge the changes from your branch into main

Navigate to GitHub.com to your repository's URL. There should be a pale yellow banner informing you about the changes you just pushed. Click the button "Compare & pull request". Notice that the title is your commit message from the previous step. Scroll down. Look at the files that have been added:

- The code is all shown in green, indicating that every line of code is new.
- The csv data file has been added but is not rendered because it is large.
- The png file is displayed.

To merge your changes into main:

- Click on the green "Create pull request" button. Github will check that it is able to merge your branch with main without problems. Note the message "This branch has no conflicts with the base branch". This means you are good to go!
- Click on the green "Merge pull request" button.
- Click on the green "Confirm merge" button.
- Click the "Delete branch" button.

### Summary

- You set up a folder on your laptop so that Git is used to track changes made *locally on your laptop*.

- You linked that folder to GitHub.com so that the same changes can be tracked externally on GitHub.
- You compared changes you made locally on your branch to the main branch on GitHub and pulled your changes into main. This means that the main branch has been updated with your changes *on GitHub*.

## 7.4 Moving forward: More practice with branching

Now suppose a few days have gone by and you are ready to work on your analysis project. In particular, you want to update some code that will affect some of the results and “outputs”, where outputs are results saved in any form. In this section, we outline the process to follow when you want to implement some set of tracked changes.

### Get setup for a new day of work

The first thing we need to do is make sure we are in a good place with git and GitHub:

- Open up Bash or Terminal and navigate to the life-expectancy folder using the `cd` command.
- Check which branch you are currently on using `git branch`. All local branch names are displayed. The asterisk is next to the branch we are currently on.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
  main
* june18-cr
```

Check if you forgot to save anything from last time using `git status`. Ideally, you have saved all your changes and there is nothing to add/track/commit. Here is my status:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june18-cr
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store

nothing added to commit but untracked files present (use "git add" to track)
```

There is one untracked file: `.DS_Store`. This is okay since it is an internal file used by Mac OS that should never be tracked. We just want to ensure no code files or outputs we intended to track have been forgotten.

You are likely still on your branch from the last day. In that case, navigate back to main using `git checkout main`.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

If you want to double check, type `git branch` to confirm you are on main.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
* main
  june18-cr
```

You can also delete the “june18-XY” branch since you no longer need to track it locally:

```
Biostat-MBP13-20:github-training corinneriddell$ git branch -d june18-cr
Deleted branch june18-cr (was 58fcc58).
```

### Pull down the changes from main

This is your **local** copy of main. It needs to pull down the changes to main that you made on GitHub in an earlier step. To do that, type `git pull origin main`. A graphic will be drawn that summarizes which files have been updated and by how much.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ 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), done.
From https://github.com/corinne-riddell/life-expectancy
 * branch           main      -> FETCH_HEAD
   884028e..5f81e34 main      -> origin/main
Updating 884028e..5f81e34
Fast-forward
  code/01_Analyze-life-expectancy.R      |  38 +
  data/Life-expectancy-by-state-long.csv | 7201 ++++++
  images/ca-black-women-LE.png          | Bin 0 -> 74550 bytes
  images/placeholder.md                |    1 +
  existing-project.Rproj              |   13 +
5 files changed, 7253 insertions(+)
create mode 100644 code/01_Analyze-life-expectancy.R
create mode 100644 data/Life-expectancy-by-state-long.csv
create mode 100644 images/ca-black-women-LE.png
create mode 100644 images/placeholder.md
create mode 100644 existing-project.Rproj
```

### Start a new branch

Like the last day, start a new branch to track today’s changes. Let’s pretend it is now June 21. Type `git checkout -b june21-XY`, where XY is replaced with your initials. Type `git branch` to confirm you have changed branches.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git checkout -b june21-cr
Switched to a new branch 'june21-cr'
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
  main
* june21-cr
```

**You are now ready to make changes!**

Think about what you would like to do in advance. In particular, suppose you want to:

- make a table that summarizes the mean life expectancy by race and gender for each state and,
- save the above table as a CSV file into the data folder.

To do so, re-launch RStudio by double-clicking the .Rproj file in your file viewer window. Navigate to the code file 01\_Analyze-life-expectancy.R and insert the following R code (if working in R) or SAS code (if working in SAS) to make and save this table:

```
# R Code
# Calculate the LE for each state, separately by race and gender:
le_averages <- le_data %>%
  group_by(state, race, sex) %>%
  summarise(mean_LE = mean(LE))

# print the first 10 rows to the screen. By default, R rounds the numeric
# information in the display to make it more compact
le_averages
# alternatively, type View(le_averages) in the Console to open up a Viewer
# window, or click the table icon beside the le_averages objects in the
# Environment pane (upper right hand panel of RStudio).

#save this table as a CSV file in the data sub-folder
write_csv(le_averages, "./data/le_averages.csv")

/*SAS Code*/

/*Calculate the LE for each state, separately by race and gender:*/
proc sort data=le_data; by state race sex; run;
proc means data=le_data; by state race sex; var le; output out=le_averages mean=mean_le;

/*Print the first 10 rows to the screen*/
proc print data=le_averages (obs=10); run;
/*Or you could just open the dataset to browse it.*/

/*Export this file to a .csv file
(if you use the following code, don't forget to replace YourFilePathHere with the appr
```

```

PROC EXPORT DATA= WORK.LE_AVERAGES
   OUTFILE= "YourFilePathHere\data\le_averages.csv"
   DBMS=CSV REPLACE;
   PUTNAMES=YES;
RUN;

```

### Track your changes using Git

Re-run your previous R code (highlight all the previous code and hit command + Return [Mac] or control + Enter [Windows]). Then run the newly-added code line by line to see what it is doing. **Don't forget to save the updated .R file by pushing the save icon!** After all that, track the changes using Git. Go to the Bash/Terminal window. Type `git status`. Which files have been modified? Which files are new and untracked?

```

Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june21-cr
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:   code/01_Analyze-life-expectancy.R

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store
    data/le_averages.csv

```

### Add new and modified files

Use `git add` to add the specific files that have been modified or created. Add them one by one. Use `git status` again to check that all the changed files are being tracked. When you are satisfied, commit these changes locally.

```

Biostat-MBP13-20:life-expectancy corinneriddell$ git add code/01_Analyze-life-expectancy.R
Biostat-MBP13-20:life-expectancy corinneriddell$ git add data/le_averages.csv
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june21-cr
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   code/01_Analyze-life-expectancy.R
    new file:   data/le_averages.csv

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store

```

### Commit the changes locally

Helpful hint: Terminal/Bash plays well with autocomplete. For example, if you are typing the pathway for the .R file as “code/...” you can push the tab button

as you are typing the name and it will autocomplete. This makes selecting the specific files to commit much easier.

`git commit -m '{your message}'` Replace ‘your message’ with a short message describing the changes. Remember to keep the quotes around the message!

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git commit -m 'calc LE averages'
[june21-cr a7435b8] calc LE averages
 2 files changed, 177 insertions(+), 1 deletion(-)
  create mode 100644 data/le_averages.csv
```

### Push the changes up to GitHub

You are now ready to push these changes up to GitHub onto GitHub’s version of your local branch. First, remind yourself of your branch’s name using `git branch`. Then push: `git push origin {YOUR-BRANCH-NAME}`, replacing `{YOUR-BRANCH-NAME}` with the name of your branch.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
  main
* june21-cr
Biostat-MBP13-20:life-expectancy corinneriddell$ git push origin june21-cr
Enumerating objects: 10, done.
Counting objects: 100% (10/10), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 2.84 KiB | 2.84 MiB/s, done.
Total 6 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
remote:
remote: Create a pull request for 'june21-cr' on GitHub by visiting:
remote:   https://github.com/corinne-riddell/life-expectancy/pull/new/june21-cr
remote:
To https://github.com/corinne-riddell/life-expectancy.git
 * [new branch]      june21-cr -> june21-cr
```

### Create a pull request

Navigate to GitHub.com and go through the steps described previously to create a pull request to pull these changes into main.

## 7.5 Another practice: Changes after receiving an updated dataset

Another week goes by. It is now June 25. You received an email that there was an error in the data file that you used to conduct the analysis. A new data file was securely transferred to you by the data holder. You need to rerun

## 7.5. ANOTHER PRACTICE: CHANGES AFTER RECEIVING AN UPDATED DATASET33

the analysis using the new dataset. The new data file is in the `future-data` subfolder of the `existing-project` that you put on your Desktop during setup.

### Get ready for the day

Set yourself up to work with Git and GitHub for the day:

`git branch`: see which branch you are on:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git branch
  main
* june21-cr
```

`git status`: confirm you committed everything you wanted to commit:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june21-cr
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store
```

nothing added to commit but untracked files present (use "git add" to track)

- `git checkout main`: switch to the main branch
- `git branch -d june21-cr`: delete the old branch (change “cr” to your initials)

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

`git pull origin main`: pull GitHub’s copy of the main branch to update your local version. Examine the figure made by Git about the changes:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ 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), done.
From https://github.com/corinne-riddell/life-expectancy
 * branch           main      -> FETCH_HEAD
   5f81e34..e15298a main      -> origin/main
Updating 5f81e34..e15298a
Fast-forward
 code/01_Analyze-life-expectancy.R |  17 ++++++
 data/le_averages.csv            | 161 ++++++++++++++++++++++++++++++++
 2 files changed, 177 insertions(+), 1 deletion(-)
 create mode 100644 data/le_averages.csv
```

### Checkout a new branch

`git checkout -b june25-xy`, replacing xy with your initials.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git checkout -b june25-cr
Switched to a new branch 'june25-cr'
```

### Update the dataset

Update the CSV file “Life-expectancy-by-state-long.csv” with the new dataset. First, decide if you want to archive this older version of the dataset for any reason. If you do, then decide where you would store the archived version and move it there. The archived version could stay on GitHub or be moved off of GitHub – this is up to you and your file organization system. Then, copy the dataset called “Life-expectancy-by-state-long.csv” from the existing-project/future-data subfolder and paste it into the life-expectancy/data subfolder.

Re-run all your code that uses this file. First open the .Rproj file to launch RStudio. Then, you can highlight all the code and hit the “Run” button.

### Track the changes using Git

Use `git status`, `git add...`, and `git commit...` to track these changes locally. Use `git push` to push these changes to GitHub.

Using `git status`, I can see that the data files have all been modified, as has the image file of the plot. There is a new untracked folder called `data/archive/` where I moved the archived dataset.

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june25-cr
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:   data/Life-expectancy-by-state-long.csv
    modified:   data/le_averages.csv
    modified:   images/ca-black-women-LE.png

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .DS_Store
    data/.DS_Store
    data/archive/
```

I use `git add` to specify all the new things I want to track:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git add data/Life-expectancy-by-state-long.csv
Biostat-MBP13-20:life-expectancy corinneriddell$ git add data/le_averages.csv
Biostat-MBP13-20:life-expectancy corinneriddell$ git add images/ca-black-women-LE.png
Biostat-MBP13-20:life-expectancy corinneriddell$ git add data/archive/
```

I then use `git status` to confirm everything is being tracked:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git status
On branch june25-cr
```

## 7.5. ANOTHER PRACTICE: CHANGES AFTER RECEIVING AN UPDATED DATASET35

Changes to be committed:

```
(use "git restore --staged <file>..." to unstage)
 modified:   data/Life-expectancy-by-state-long.csv
 new file:   data/archive/Life-expectancy-by-state-long_old.csv
 modified:   data/le_averages.csv
 modified:   images/ca-black-women-LE.png
```

Untracked files:

```
(use "git add <file>..." to include in what will be committed)
 .DS_Store
 data/.DS_Store
```

### Commit the changes locally

Then I commit my changes:

```
Biostat-MBP13-20:life-expectancy corinneriddell$ git commit -m 'data update and downstream changes
[june25-cr c4b02db] data update and downstream changes
 4 files changed, 7570 insertions(+), 369 deletions(-)
  create mode 100644 data/archive/Life-expectancy-by-state-long_old.csv
  rewrite images/ca-black-women-LE.png (98%)'
```

### Push the changes to GitHub

Finally I push these changes to GitHub:

```
git push origin june25-cr
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 8 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (8/8), 68.08 KiB | 9.73 MiB/s, done.
Total 8 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
To https://github.com/corinne-riddell/life-expectancy.git
 e15298a..c4b02db june25-cr -> june25-cr
Biostat-MBP13-20:life-expectancy corinneriddell$
```

### Submit a pull request

Go through the process to start a pull request. On the pull request page, scroll down to see the “diffs” in the data and image. Pay close attention to the files that were changed:

- use the “2-up” “swiper” and “onion skin” tools to see the changes to the saved figure. How did the data changed? Which tool do you prefer?
- can you see which rows of data were affected by the change?
- can you tell from the data or images how the change affected the analytic findings?
- anything else you noticed?

After looking at the changes, go through the process to merge the pull request into main.

## 7.6 Summary

Congratulations, you have successfully worked in Git and GitHub solo! What a journey. You now know how to transition your existing work to being tracked using Git and GitHub, and how to track new changes using `git add`, `git commit`, and push them to your GitHub using `git push origin {branch}`. You also learned how to pull these changes back into main using `git pull origin main`, and how to create, switch, and delete branches.

There are a lot of steps to remember, but if you do this for a week it will become easier. To help you get started, feel free to use the “Daily Git Practice” guide (linked on the left), which provides a handy guide covering all the commands covered in this section.

# Chapter 8

## Group Workflow

### 8.1 Outline for this section

In the previous section, you got up and running with working alone in Git and GitHub. During this section, we're going to learn how to collaborate with others using Git and GitHub. We will:

- Set up a shared repository
- Make a first commit
- Cover three routes for collaboration:
  - Independent Work
  - Code Review
  - Working on the same files
    - \* Practice resolving a merge conflict

### 8.2 Setup

To get the most out of this section of the training, you need a training buddy. One of you will be the *lead researcher*, and the other will be the *colleague*. The lead researcher will create the repository, and the colleague will contribute to it.

#### **Lead Researcher: Create a new repository on GitHub**

- Go to GitHub, click on the “+” icon in the top-right corner, and select **New repository**.
- Follow the same instructions as we did in the solo workflow to create a new repository (add a README, choose a license, etc.).
- Add your colleague as a collaborator by going to **Settings > Collaborators > Add people**. Ask your colleague for their GitHub username and add them to this repository.

### Colleague: Join the repository

- Check your email for an invitation to collaborate on the repository.
- Accept the invitation by clicking on the link in the email.
- Star this repository by clicking **Star** in the top right - this will make it easier to find it later if you navigate away!

### Both: Clone the repository

- Go to the main page of the repository clicking on the green **Code** button and copying the URL. Then, in Bash/Terminal navigate to the location where you will clone your repository and run the following command:

```
git clone {repository URL}
```

## 8.3 Get oriented to the new directory

- Navigate into your repository by typing `cd {repo-name}`
- Type `git status`. The results shows you that no changes have been made yet:

```
Biostat-MBP13-20:collabo-repo corinneriddell$ git status
On branch main
Your branch is up to date with 'origin/main'.
```

```
nothing to commit, working tree clean
```

- Type `git branch`. This shows you that you are currently on the main branch.

```
Biostat-MBP13-20:life-expectancy corinneriddell$git branch
* main
```

- **Both:** Create a new branch. Now that we know that our repository is cloned, up to date, and on main, we can make a new branch to work on:

```
git checkout -b {new-branch-name}
git branch
```

Typing `git branch` will show you that you are now on the new branch, though only the first line was *necessary* to create this branch.

## 8.4 Collaborative Routes

In general, there are three ways to work collaboratively on a project:

1. **Collaborative route #1 (Independent work):** You each work on separate files and merge your own changes. It is similar to creating several separate documents and storing them in a shared folder for you

both to view.

2. **Collaborative route #2 (Code review):** Submit pull requests for code reviews without direct edits. It is similar to having a shared document where you can leave comments for each other - like when you leave comments on a MS Word document for your colleague.
3. **Collaborative route #3 (Working on the same files):** You both work on the same file and merge in your changes. Sometimes this leads to merge conflicts so additional care is needed. It is similar to having a shared document where you can make changes directly to your colleague's work – think about working collaboratively on a document in the cloud where both individuals can edit the document at the same time, but weird things might happen if you are working on the same sentence at the same time.

We will cover each collaborative route below.

#### 8.4.1 Collaborative Route #1: Independent work

You each work on separate files and merge your own changes. To start, the repository will have two files, one that each of you made. In general, you could maintain several files, each of you working on the specific files that make sense but never editing each other's files.

Let's practice this workflow now:

- **Lead researcher:** Save a code file that you brought to this workshop in the shared repository.
- **Colleague:** Save a code file that you brought to this workshop in the shared repository. If your file has the same name as the lead researcher's file, rename it now.
- **Both:** Add, commit, and push your respective files to GitHub:

```
git add {file-name}  
git commit -m "Initial commit with {file-name}"  
git push origin {new-branch-name}
```

If the output from the push command looks successful, you are now ready to create a pull request following the same process we used in the solo workflow section.

- **Both:** Navigate to GitHub.com to your repository's URL. There should be a pale yellow banner informing you about the changes you just pushed. Click the button “Compare & pull request”. Notice that the title is your commit message from the previous step. Scroll down. Look at the files that have been added.
- The code is all shown in green, indicating that every line of code is new.
- Merge the changes from your branch into main

- Click on the green “Create pull request” button. Github will check that it is able to merge your branch with main without problems. Note the message “This branch has no conflicts with the base branch”. This means you are good to go!
- Click on the green “Merge pull request” button.
- Click on the green “Confirm merge” button.
- Click the “Delete branch” button.

Both of your files have been merged into the main branch. Let’s take a look. Go into the main `Code` page and take a look at the code on `main` (which should be the only existing branch in your repository). You should see the files that *both* of you created in the previous steps.

### Recap

- You have now successfully completed the first collaborative route. You each created a file, committed it, and merged it into the main branch. You worked on different files and had no merge conflicts or interaction, but now have access to each other’s work!
- In this workflow, you could continue working independently on your file in the shared repo and add other files as you go along.
- Take care: Now that you are working together, be sure to use `git pull origin main` to frequently grab any changes your colleague made to the repository. You should be doing this already, but it is even more important when you are working together so you can stay up to date on whatever changes have been made to the repository.

#### 8.4.2 Collaborative Route #2: Code Review

In Collaborative Route #2, the lead researcher will make changes to a file and push these to GitHub. The colleague will then review these changes by submitting comments on the pull request.

Let’s practice this workflow now:

##### **Everyone:**

- Checkout main, and pull down the changes

```
git checkout main
git pull origin main
git status
```

##### **Lead researcher steps:**

- Create a new branch. Pretend it is the next day and include this date in the branch name alongside your initials:

```
git checkout -b {new-branch-name}
git branch
```

- Make and **save a change** to the file you added in the previous step. You can add in a comment or a new line of code - something small is good. **Save your file.**
- Add, commit, and push your changes to your branch on GitHub:

```
git status
git add {file-name}
git commit -m "{your description of what you changed}"
git push origin {your-branch-name}
```

- Go to your GitHub repository page. You should see a notification about your recent push. Click on “Compare & pull request” next to your branch.
- **NEW STEP ALERT:** Fill in the details of the pull request, explaining what you’ve added or changed and why.
- **NEW STEP ALERT:** Add your colleague as a reviewer. We will not immediately merge this in this time!
- Submit the pull request.

#### Colleague steps:

- Once the pull request is submitted, the colleague should receive an email telling them that the lead researcher requested their review on PR #X.
- Click **View it on GitHub** and it will bring you to GitHub and there will be a yellow box at the top.
- Click on **Add your review** in the yellow box. Then click **Review changes** in green in the right corner.
- Leave comments on the pull request by clicking the plus signs next to chunks of code. You can leave multiple comments. Your feedback can include suggestions, questions, or general feedback about the code. Focus on clarity, efficiency, and any potential errors you might notice.
- When you’re finished commenting, click **Finish your review** in green in the top right.
- Submit your comments. You have three review options:

Let’s take a moment to talk briefly about each of these review options.

---

<b>Comment</b>	This submits feedback without explicitly approving the pull request. This will send the lead researcher an email with your comments on their code. The lead researcher can click <b>View it on GitHub</b> to see your comments. They can reply and/or merge the pull request into main at this point. This can be used either as an FYI or to start a conversation about the code - e.g. maybe a certain function is new to you so you want to ask about why they chose to use that. You may want to use this if you have a workflow where you always want the person who wrote the code to do the merging.
----------------	---

---

<b>Ap-</b>	This submits feedback and approves the PR for merging into main.
<b>prove</b>	This will send the lead researcher an email with your comments that will also indicate that the pull request is approved. This option can be used as an FYI, e.g., telling them about an alternate function that they <i>could</i> use but do not have to. Either person can merge the pull request into main. You can establish norms around how these work with your colleagues - there is no right answer!
<b>Re-</b>	Submit feedback that the other person needs to address before they
<b>quest</b>	can merge into main. This will send the lead researcher an email
<b>Changes</b>	with your comments that will also indicate that the PR needs changes. This option can be used if you see something that needs to change before the pull request can be merged - e.g. a function that is not working as intended. The lead researcher will need to address your comments before they can merge the pull request into main. <i>Technically, the PR could be merged in without the changes, but this is not good practice.</i>

---

**Lead Researcher:**

- Review the comments made by your colleague. Make any necessary changes to your code based on the feedback.
- Add, commit, and push these changes to the same branch. This will automatically update the pull request.

**Colleague:**

- You will receive an email saying that the lead researcher pushed one commit. At this point, the colleague can click `View it on GitHub` and verify that the changes are sufficient.
- Review the changes made by the lead researcher in response to your comments. If you're satisfied with the updates, approve the pull request.

**Lead Researcher:**

- Once the pull request is approved, merge your changes into the main branch.

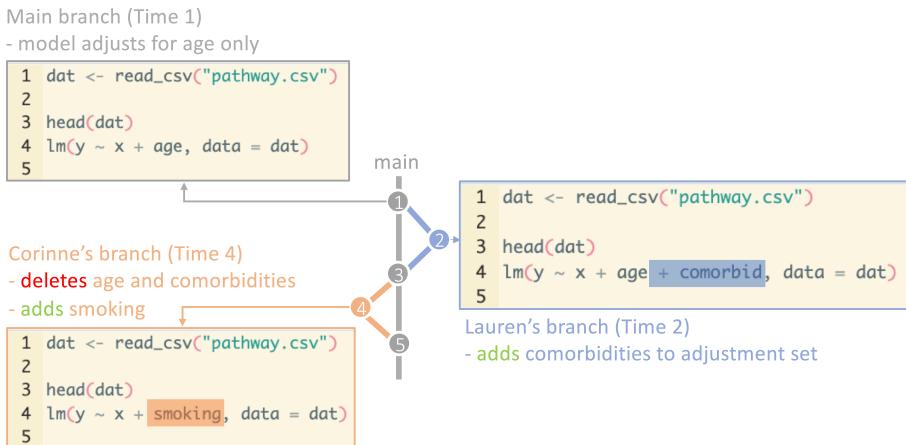
**Recap**

- You have successfully completed the second collaborative route. You used Git to review/check/comment on your colleague's work without emailing or copying/pasting code back and forth. The lead researcher made changes to their code based on your feedback, and you both worked together to finalize the pull request.
- Similar to the previous workflow, collaborators can work independently, but request review of each other's work.

### 8.4.3 Collaborative Route #3: Working on the same files

In this scenario, imagine that the lead researcher has already created and committed an analysis file. The colleague then wants to make direct changes to that file or extend the analysis. This is a commonly used workflow, and works well if both people are working sequentially and pulling from main often to pull down any changes made to the repo as their collaborators work.

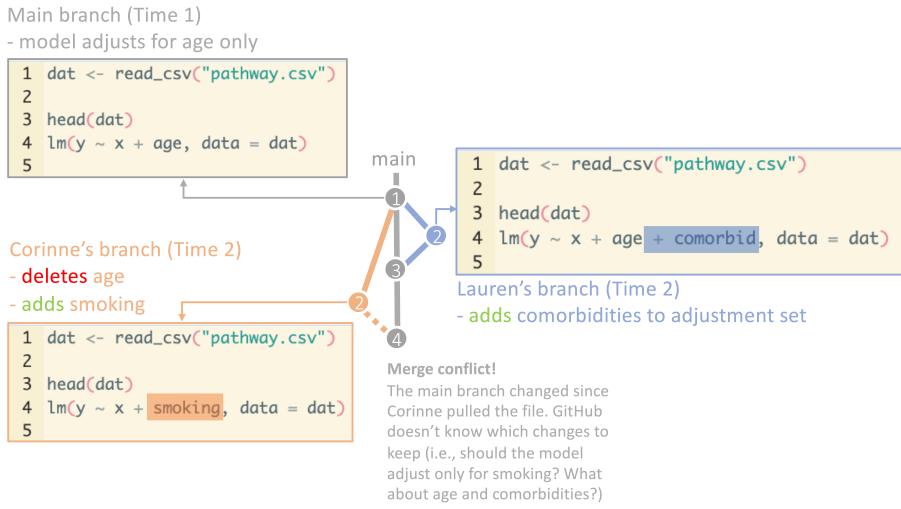
This diagram shows how both Lauren and Corinne can work on the same file, sequentially over time, each pulling from main, making changes, and merging their changes back into main before the other person works:



However, this collaborative route leaves room for the dreaded merge conflict.

Merge conflicts can be a source of stress for many GitHub users. While they can be scary, you are in good company when you have your first merge conflict – everyone has a merge conflict eventually.

What is a merge conflict, you might ask, and how does it happen? Well, a merge conflict can happen if you and a colleague both pull from main at the same time and proceed to edit the same location on the same file. If colleague #1 merges their changes into main, then when colleague #2 goes to merge their changes into main, GitHub doesn't know what changes to keep.



If this seems a little convoluted, it is your lucky day. We will set the stage for a merge conflict and practice resolving it with our buddy.

### Creating a merge conflict

#### **Everyone:**

- Checkout and pull any changes: `git checkout main` and `git pull origin main` to make sure you are up to date.
- Make and checkout a new branch: `git checkout -b {new-branch-name}`. Each of you should call their branch something different from the other researcher.
- Together, choose one file and one line location on the file that you will each edit separately on your laptops.
- Each of you should make the edits you agreed upon to the single line location that you decided you would edit. You should both make *different* changes.  
**Save your code file..**

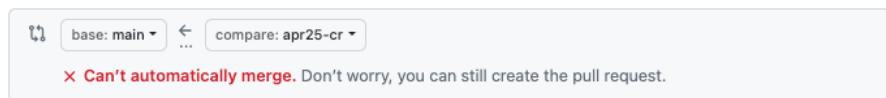
- **Lead researcher only:** add, commit, and push your changes to your branch. Then, create a pull request and merge your changes into main.
- **Colleague:** Your turn. Add, commit, and push your changes to your branch. Then, create a pull request. Merge conflict! You will not be able to merge into main.

Deep breaths. Let's walk through how to resolve the merge conflict.

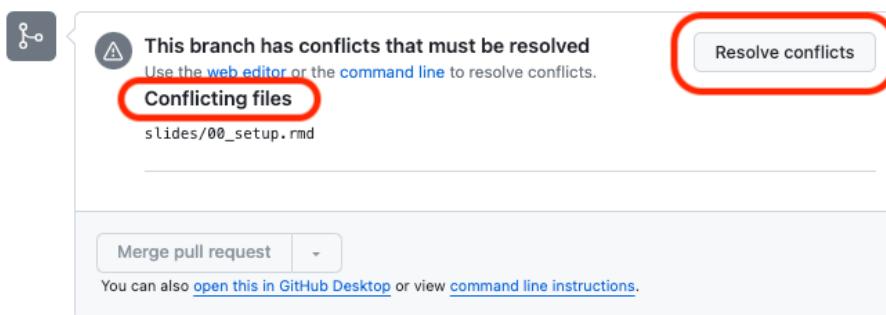
The colleague with the conflict will see the following. Colleague, follow these steps! Lead researcher, watch your colleague's screen to follow along:

### Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff comparisons here.



Click on `View Pull Request` to begin resolving the merge conflict. When you click on `View Pull Request`, you will see the following:



Click on `Resolve conflicts` to begin resolving the merge conflict. When you click on `Resolve conflicts`, you will see the following:

```

20
21 <<<<< conflicted_branch_name
22 Anything displayed after the "<<<<<" and before the "===="
23 is the code suggested on the branch "conflicted_branch_name"
24 =====
25
26 Anything displayed after the "===== and before the ">>>>>" 
27 is the code that was suggested by your collaborator that was
28 previously merged into main
29 >>>>> main

GitHub-added delineators
between person #1 and
person #2's code
  
```

Together, look at the code in the file. Decide which edits you will keep. For this exercise, it is arbitrary which edits you keep. However, in practice you will make this decision in an informed way.

When you choose what to keep, delete the `>>>` and `==` and `<<<` lines, as well as the lines from the person whose code you are not keeping. When you are

finished, click `Mark as resolved` in the top right corner. Now that you have clicked `Mark as resolved`, you will see a green check mark next to the file name, and it will say `Resolved` in the top right corner. There will also be a green button in the top right corner that says `Commit merge`. Click on this button. This brings you back to the page you are familiar with, where you can merge into main. Click `Merge pull request` and then `Confirm merge`. You can delete the branch as you usually do.

**Everyone:** You can both checkout main and pull down the changes. You will now see that the file with the merge conflict has been cleaned up and you will see only the code that you accepted above!

### Recap

- You have just resolved your first merge conflict!
- In this route you worked together on the same files. If you had edited different sections of the file, you could successfully merge both of your changes into main. In this instance, both of you edited the same line of code, which created a merge conflict. You resolve the merge conflict.
- To minimize the chance of merge conflicts, make sure you do `git pull origin main` before you start your day's work to pull down the most recent version of the repo. If you do get a conflict, you know how to resolve it. You will need to make a decision about which version of the code to keep. It should be clear based on what you hope to accomplish in the analysis.

# Chapter 9

## What to track on Git and Github?

### 9.1 Approach 1 : Track statistical code only

Some people use Git and Github to track changes to their statistical code only. Here, they are only tracking their code files in R (.R or .Rmd), or SAS (.SAS), or the corresponding files for whichever language they use.

They add all other types of files to the .gitignore file so that Git won't track them or accidentally push them to GitHub. Under this approach, untracked files include: data files, word documents, PDFs, image files, etc – anything else you are storing looking in the folder tracked by git that you do not want pushed to GitHub.

This approach most closely corresponds to how GitHub is used by software developers/other code writers, and is used by researchers and research teams depending on their objectives.

### 9.2 Approach 2: Track statistical code, some data files, images, and other documents

Some people use Git and Github to track a variety of files including their code, *some* of their data files, as well as images and other documents. This approach is often used by advocates of reproducible research, and to give outsiders a fuller picture of an entire research project.

Sometimes, you can post your data, if it is unrestricted and not too large. In this case, you can set up your GitHub so that all of your data visualizations as

well as a completed report can be reproduced by someone who is not affiliated with your project.

In many research or public health settings, you are not able to share the data. In this case, you can still share other aspects of your project with the wider community.

### 9.2.1 Data files do's and dont's

#### **Don't track restricted data!**

Restricted data cannot be posted on GitHub. A best practice is to store it securely outside of your tracked folder to ensure there is no chance of it being posted.

#### **Don't track large datasets!**

Git will warn you if your file exceeds 50 MB and block you from tracking files 100 MB or larger. To ensure these files aren't tracked, you can store them elsewhere (outside of the tracked Git folder), or store them in the tracked folder while also listing them or their file type to your .gitignore file.

#### **WARNING FOR MAC USERS: UNDER NO CIRCUMSTANCES SHOULD YOU TRACK .DS\_STORE files!**

If you are a Mac user, you may not even know what a .DS\_STORE file is. These are invisible files that are created whenever you view a folder using Finder. You will see them when you write `git status`. You do not want to track these files as it can allow people to hack your computer!! As a best practice, add these to your .gitignore file to ensure that they are never tracked by you or anyone else contributing to your project.

#### **Consider tracking intermediate data products**

Generally, it also doesn't often make sense to track a large "raw" data file – it is too big and not useful to track any changes to this file. It may be helpful to track "intermediate data products", if these files are not restricted. Intermediate data products might include aggregated datasets that are either reported directly or used in analysis. For example, you may have access to a restricted dataset, but the data may become non-restricted if you aggregate at the level of the census tract. The benefit of tracking this smaller dataset is that if the raw data is updated, you can easily see how those updates affect these intermediate data products if you track them. In this case you need to ensure you are not reporting any private/restricted data (eg no cell counts below 10 is a restricted often imposed on aggregated tables, or not reporting any identifying features such as protected health information or anything else that would allow anyone with access to identify individuals.

#### **Do track plain text data files (e.g., csv and txt files)**

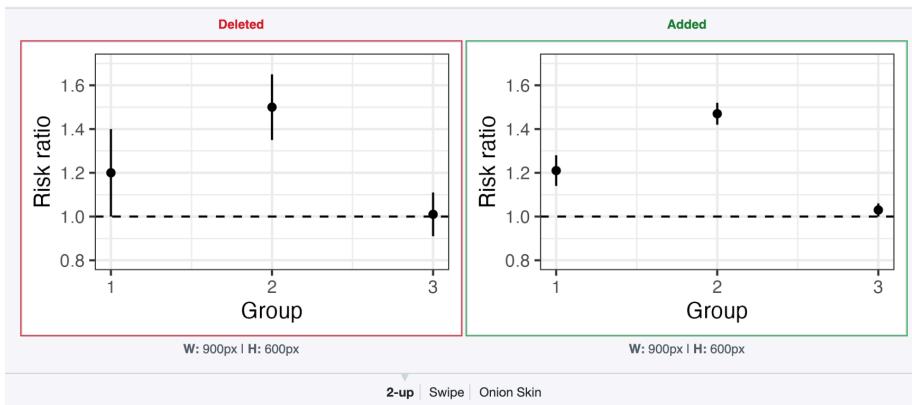
## 9.2. APPROACH 2: TRACK STATISTICAL CODE, SOME DATA FILES, IMAGES, AND OTHER DOCUMENTS

These are best for tracking because they render nicely on GitHub, so you can easily view the differences to these files when you submit a pull request. You could also track things like Excel files but you can't easily view them on GitHub, so some of the benefits of using GitHub do not apply to these files.

### 9.2.2 Images dos and dont's

You can also track image files (e.g., png, jpeg), such as plots/other figures you create for a report. The benefit of tracking figures are the use of the image comparison tools in GitHub to see an image pre/post a change in the analysis. This can be super helpful when you have modified something in the analysis after having already written up some results. If you track the image, you can easily see how it changed (it being the point estimates and confidence intervals, of the slope of the regression line, or the shading of a colored map). This has the direct pay-off of making it much easier to revise the written result.

#### 2-up comparison:



#### Swipe comparison:

#### Onion skin comparison:

#### Dont' track very large image files

One thing to be careful about is not uploading very large images like the ones that are generated by some GIS analyses (large maps). If you aren't sure if you should track your image file, take a look at the file size. (I had a look at all of my repositories for epidemiologic analyses – most images were < 1000 kb. Some were between 1 MB and 4 MB – these were some maps and some images saved at higher resolutions.)

### 9.2.3 Documents

You can also track reports and manuscripts using GitHub. If these reports are written in a plain text language (e.g., R markdown, LaTeX) then they will

render nicely on GitHub and, and permit you to see the “diffs” made to the document during a pull request. Tracking pdfs is permissible but you can’t easily see the “diffs” when they are updated. However, pdfs are easily viewable on GitHub. Tracking docx files is also permissible, but you can’t easily see the “diffs” and they are not easily viewable on GitHub (since they require MS Word to render them).

**Do track plain text reports like R markdown files**

**Consider tracking PDFs (easily viewable on GitHub)**

**Don’t track Word documents (cannot view)**

### 9.3 Things you definitely do not want to track

**Anything that is private or restricted or that you would never want to inadvertently share.**

- This includes passwords, or API keys that you might use to extract data.
- For example, I have an API key to access Census data. I do not include this API key in my GitHub repository. There are methods for writing the code to still access the key on my personal computer without writing it out explicitly in the code.

# Chapter 10

## Daily Git Practice

Use these instructions each day you are working in your repository for good practice!

1. Open Terminal (Mac) or Bash (Windows). Navigate to your repository using `cd`.
2. When you get to your directory, type `git branch` to remind yourself which branch you are on as you main be on a branch from the previous work day. Type `git status` to ensure you committed and pushed everything the last time you used Git and GitHub. If there is nothing to commit, go to the next step. If there are files with changes to be committed, you will need to figure out if you want to commit those changes or if you want to discard them. If you want to discard them, type `git restore {file-name}` to discard changes to that file. Do this for all files until you have a clean directory if you are sure you don't want to track those changes. If you want to commit them, follow steps 10 through 19 before proceeding to Step 2.
3. Type `git checkout main`.
4. Type `git pull origin main`. This will pull the changes you made the day before and get you ready to start on a clean branch today.
5. Delete the previous day's branch. Type `git branch -d {branch-name}`. Remember to replace `{branch-name}` with your previous day's branch name. If you don't remember what it was named, type `git branch` to display a list of all the branches you have.
6. Create and checkout a new branch by typing `git checkout -b {new-branch-name}`. Replace the `{new-branch-name}` with a new branch name. This may be something like `{your-initials_date}`.
7. Do your work. Try to perform "your work" as a discrete task that you can summarize with one commit message. If you have more to do, then make multiple commits in one day.
8. Once you are done, go back to Terminal/Bash (where you should already

be in your repository directory) type `git status` to see which files have been modified and should be staged for a commit. You may have some untracked files that you created and also want to track.

9. Add your newly added and modified files to the staging area by typing `git add {file-names}`. You can add all the files that you edited/created.
10. Commit your changes by typing `git commit -m "{commit-message}"`. Replace `{commit-message}` with your commit message (don't forget to keep the quotes around the message)!
11. Remind yourself of your branch's name by typing `git branch`. Then push your changes to the remote repository by typing `git push origin {new-branch-name}`. Replace `{new-branch-name}` with your new branch name.
12. In a browser, go to your repository on GitHub.
13. If this is the first time you committed to this branch, there should be a yellow banner on your GitHub home page with a “pull request” button. If not, you can navigate to the Code tab, change to your branch, and you will see the code that you just worked on! It is now up on GitHub!
14. Click on the `Pull requests` tab.
15. Click on the green `New pull request` button.
16. Click on the green `Create pull request` button.
17. Click on the green `Merge pull request` button.
18. Click on the green `Confirm merge` button.
19. Delete your branch on GitHub when prompted.
20. You are now ready to do this all over the next day!

# Chapter 11

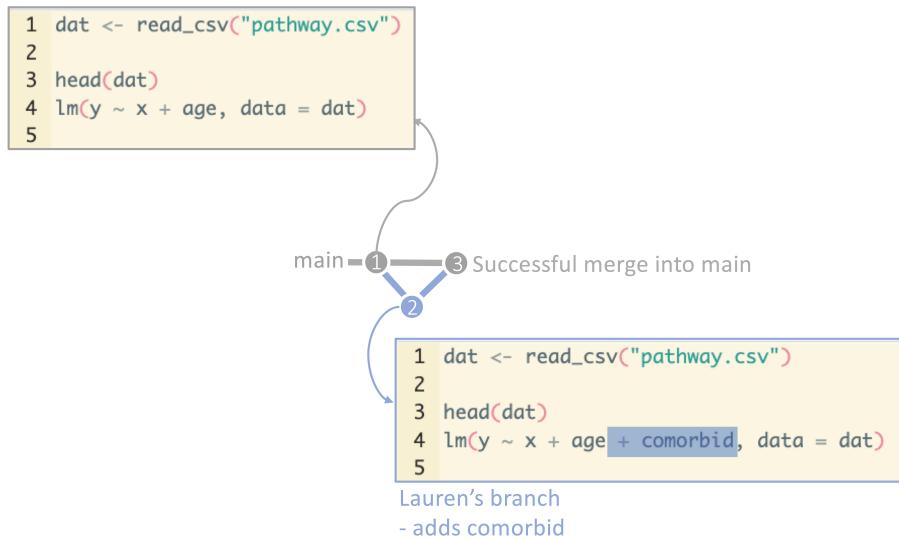
## Assorted Git processes and tips

This page introduces a few Git and GitHub processes that didn't nicely fold into the other chapters, but which we think make be useful to you

### 11.1 Forking

Remember when we went through **branching?** **Branching** is the process of creating a new branch where you can make changes to code separately from the main branch. When you finish your changes, you merge your branch into main and delete the branch. **Branching** is a critical tool for working with Git repositories and ensuring that the main branch remains functional and clean. Even if you do not merge your branch back in, the owner of the repository can see your branch on GitHub, take a look at what you have done, and foster collaboration with the members of the repository.

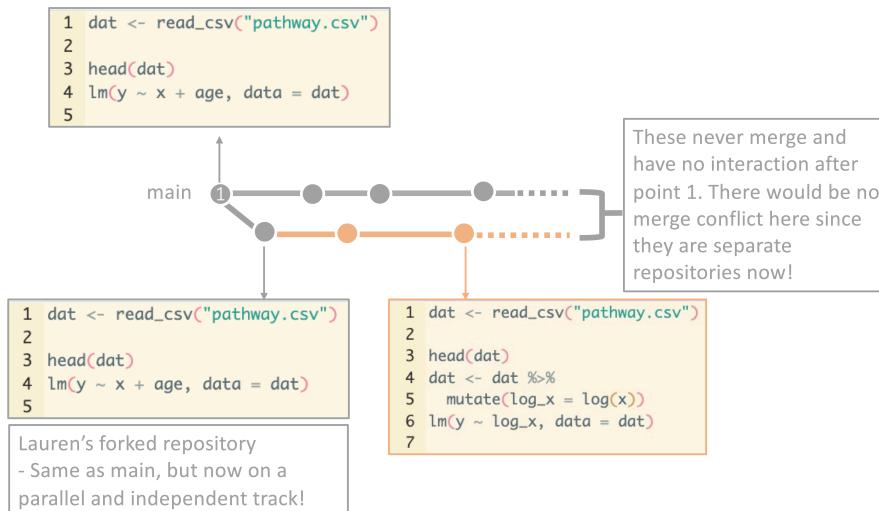
**Branching:**



**Forking** is a similar process, but it is used when you want to build off of someone else’s repository. When you **fork** a repository, you create a copy of the repository in your GitHub account. You can then make changes to the repository in your account without affecting the original repository. The owner of the original repository will not be a member of your forked repository. The forked repository is essentially your repository now. You can think about this as similar to receiving a Google folder, downloading its contents, and saving it to your computer. The original Google folder is still there, the people with ownership and access to the original Google folder are unaffected, but now you have a completely separate folder on your computer that is not a collaboration with the original Google folder team in any way.

Why fork? Forking is a great way to contribute to open-source projects or to change a repository that you don’t have “write” access to. Forking also circumvents reinventing the wheel – perhaps your colleagues developed a suite of functions for cleaning malaria data, and you need similar functions for your dengue project. You can fork their repository, make changes to tailor these functions to dengue, and use their functions in your project.

### Forking:



**Forking** is core to the premise of Git and the effort to move towards open-source software development. It allows for collaboration within workplaces, disciplines, and around the world for the sharing of code and ideas.

## 11.2 Ignoring files

When you are working on a project, you will often have files that you do not want to track with Git. Such files, such as restricted datasets and large datasets, were discussed in the “What to track on Git and GitHub” chapter.

Decisions around data storage are a team decision and may require input from your information technology department or institutional review board. If permitted according to the data’s level of security, some teams will store some data on Git in a private repository, while other teams will use shared drives for data storage and Git/GitHub for code sharing, and, finally, other teams will store the data in their Git repository but not push them to GitHub.

About that last one – yes, you can store files in your GitHub folder but never choose to track them. To do this, use a `.gitignore` file to tell Git to ignore these files.

A `.gitignore` file is a list of files that Git is instructed not to track. Whenever you add files in your repository, for example, Git will ignore the files listed in your `.gitignore` file.

Here’s an example of the contents of a `.gitignore` file:

```
1 # SQL db
2 *.sqlite
3 *.sql
4
5 # Builds
6 build/
7
```

This `.gitignore` file instructs Git to ignore any files that end with `.sql` or `.sqlite`, as well as any files in the `build` subfolder. Git ignore files are handy if you have a workflow where you store most of a project's files within the Git repository, since it allows you to specify which files to ignore. Git ignore files are less useful if you have a workflow where you want everything in your repository to be pushed to GitHub.

### 11.3 Rolling back changes

Another advantage of Git and GitHub is the ability to roll back changes. Perhaps you committed a few iterations of code and realized that the code you wrote three commits ago was, in hindsight, the best version. No worries – you can roll back to that commit and start from there. This is a great way to experiment with code and try new things without worrying about losing your work. To do this, you can either use `git checkout` to check out a previous commit, or you can use `git revert` to revert a commit. This process is a bit complicated, but there is a great walk through here.

### 11.4 And if all else fails...

Remember that Git is a tool that can help you track versions, examine changes, and collaborate with others. If you get stuck, never forget that it's okay to delete your repository and start over - both of these website's co-author have done this (repeatedly... even while creating this website).

There are many comics and memes about Git and the struggles that come with it – include the acceptability of burning down your repository and starting over!

Remember that you are not alone in your struggles and that there are many resources available to help you. It is **ALWAYS** okay to just save your code elsewhere for a moment and start over.