An Introduction to Working with Git, GitHub, and RStudio

1 What's the Point?

The point of GitHub is version control. GitHub is useful for many things, but the platform was designed with the express purpose of providing effective version control for collaborative coding projects, so that is what it's best at.

It's useful (and accurate) to think of GitHub having three different parts: the remote repository, the staging area, and the local repository. Repositories are folders dedicated to a single project.

- The remote repository is hosted by GitHub; it's kept on their servers. For us, the users, the remote repository is where we keep the latest, agreed upon, up-to-date code. Everyone can see this code.
- The local repository is hosted on our own computer. It is essentially a copy the remote repository where we can make any changes we like without affecting the code on the remote repository. Only the local user can see this code.
- The staging area is where we put changes we made to the code in our local repository that we think should be made officially to the remote repository. When code is in the staging, everyone can see the code, particularly the changes made. We can then agree on the changes that should officially be made to the remote repository.

The above is the basic workflow of GitHub. First, we copy the remote repository to our local machines. Then we make any changes we think are necessary. Changes are submitted to the staging area. Finally, we decide which changes to keep and update the remote repository.

Whenever changes are made, we document it by including a message. We can then review the changes and messages to get a history of how the code has developed.

1.1 Git v. GitHub

It may seem as if I'm using the two interchangeably, but they're not actually the same. Git is the language and source control program, while GitHub is the cloud service where we store code. We use git commands to interact with GitHub. The distinction is not terribly important, but hopefully it will clear up any confusion about why I switch back and forth between the two as we go.

2 Installing Git

Before using git, we have to install it on our machines.

For Mac :

1. Open your terminal. To do this, hit the command key and space bar and start typing terminal. When the terminal appears as the highlighted option, hit return. This will open your terminal.

2. If you don't have Homebrew installed, it's time to install it. Homebrew is a package that simplifies package installations in Mac. If you think you already have it, you can skip to the next step and see if it it works. If the next step doesn't work, come back here. To install Homebrew, simply copy and paste the following code into your terminal and hit return:

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

3. Now we can install git. Again, copy and paste the following code into your terminal and hit return:

   ```
   brew install git
   ```

4. Git is installed!

For Windows :

1. Click here to navigate to the git homepage.
2. In the downloads box, click the Windows options. Git for Windows will automatically begin downloading.
3. Go to where the installer downloaded, and double click to start installing.
4. I recommend keeping all the default settings EXCEPT in potentially one place: your default text editor. The default in Windows is Notepad++, but if you would rather use something else (e.g., SublimeText, Sweave, vim, emacs, etc.), this is the place to change it. For everything else, it's best to stick to the defaults.
5. Once you hit finish, git is installed!

3 Terminal and Command Line Basics

The terminal is how you talk to your computer. The command line is where you speak (type). Git is a language for the command line. It is possible to use git and GitHub with a **g**raphical **u**ser **i**nterface (a GUI), but it's not worth learning. Despite any qualms or trepidation feelings you might have, the most straightforward way to use git is with the command line in your terminal. There are four general, non-git commands to know first.

`pwd` This is how you find your current working directory.

`ls` Show everything in the directory.

`cd DirectoryName` Change your directory; moves to the directory "directory_name." To go "backwards", i.e., move to the directory above the one you're in, is `cd ..`

`mkdir DirectoryName` Create a new directory called "directory_name"

These commands help you navigate around your computer. The working directory is where your code is running and (if you don't specify otherwise) where anything you save is kept. Directories are like folders used to organize your computer. For example, within your `Documents` folder, you might have two subfolders, `Literature` and `WorkingPapers`[1]. Say `Literature` has all the articles you've downloaded in it and `WorkingPapers` is where you save all the articles you're writing that will eventually be accepted at AJPS. `Documents` is a working directory, but both `Literature` and `WorkingPapers` can be as well.

The following examples will demonstrate how to use the above four commands to see where you are, see what is in your folder/directory, and how to change between them. The symbols `~%` indicate the start of the terminal line where we begin typing. The user (we) do **not** need to type these symbols or anything prior to them. What we type is in burnt orange[2]. Key words are bold-ed. Lines without `~%` are what the computer returns. Comments are in slate gray; these are for additional clarification and should **not** be typed out by the user. The examples will go over what we type in the command line and what the computer will return.

3.1 Examples of Command Line Basics

- Checking which directory you are currently in:

```
username@computername ~ % pwd This is an example comment; do not type words in slate gray
/Users/username  An example of what the computer might return
```

- Changing into the `Documents` directory and checking to see where we are now:

```
username@computername ~ % cd Documents
username@computername Documents ~ % pwd
/Users/username/Documents
```

- Looking to see what's inside the `Documents` directory:

```
username@computername Documents ~ % ls
Literature WorkingPapers
```

- Changing directories into the `Literature` directory:

```
username@computername Documents ~ % cd Literature
username@computername Literature ~ % pwd
/Users/username/Documents/Literature
```

- Now say we want to go into our `WorkingPapers` directory. First, we need to go back to the `Documents` directory. Then, we can go into the `WorkingPapers` directory.

[1]If you're going to be working from the terminal, it's good practice to make filenames without spaces. Spaces have meaning in the command line, so if the name of you file you want to call from the command line has spaces in it, the terminal will get confused.

[2]Burnt orange is defined at this useful website.

```
username@computername Literature ˜ % cd ..
username@computername Documents ˜ % cd WorkingPapers
username@computername WorkingPapers ˜ % pwd
/Users/username/Documents/WorkingPapers
```

- We can actually do the above in one step. I'll demonstrate this by switching back to the `Literature` folder from the `WorkingPapers` folder.

```
username@computername WorkingPapers ˜ % cd ../Literature
username@computername Literature ˜ % pwd
/Users/username/Documents/Literature
```

- Finally, say you want to create a new folder in `Literature` just for articles relating to MENA called `MENApapers`.

```
username@computername Literature ˜ % mkdir MENApapers
username@computername Literature ˜ % ls
MENApapers
```

- Bonus: you can look inside a directory you're not currently in, as long as you tell your computer where to look. For example, if we were back in the `Documents` directory, we could ask the computer what's in the `Literature` directory.

```
username@computername Literature ˜ % cd ..
username@computername Documents ˜ % ls Literature/
MENApapers
```

4 Git and RStudio

A note on jargon: So far I've talked a lot about directories and folders. They can essentially be used interchangeably. In git-lingo they're called *repositories*. This makes sense, because you are storing, or repositing, your code on GitHub. The place you reposit your code is a repository. In practice, you navigate repositories just like directories. Cool kids call them "repos."

4.1 Git Setup

Now that we're all comfortable using the command line, we can get started with git. The first thing we want to do is set our global environments. We do this so whenever we checkout a repository[3] or submit changes to a file, our name is associated with it. Then, anyone can see who changed what. The global environments we want to set are our user name and email. The commands to do this follow:

```
˜ % git config --global user.name "User Name" Enter your own user name in quotes
˜ % git config --global user.email "user.email@princeton.edu" Enter the email associated with your
    GitHub account
```

[3]Information on this coming soon...

4.2 Git and GitHub for Personal Use

Step 0. Deciding Where to Put Things Whenever we create anything– papers, code, to-do lists, etc.– it's important to think about where we want to save that thing so we can easily find it later. I recommend having a folder dedicated to GitHub projects. That folder can be your main working directory whenever you're working with git. We can use to opportunity to practice making a new directory from the command line. The following commands create a directory called `GitHub` and then put you in that working directory.

```
username@computername ~ % mkdir GitHub Making a GitHub directory
username@computername ~ % cd GitHub Changing into that directory to make it your current working directory
username@computername GitHub ~ % pwd Confirming your current working directory
/Users/username/GitHub
```

From this point forward, I'm going to assume you are in the working directory you want to be in and not include anything before `~%` in the examples.

Step 1. Creating a Repository Before we can work in or on a repository, we need to create one. There are two ways to go about this: (1) create a remote repository from scratch or (2) turn a local project folder into a remote repository. For now, let's just concentrate on the first method. We can go over the second method later.

Creating a Remote Repository from Scratch :

1. Navigate to your GitHub page online and click on the Repositories tab.
2. On the right, you will see a green button that says "New." Click on it.
3. Give your new repository a name. For the sake of this tutorial, I suggest `Trying_Out_Git`.
4. Check the box for "Add a README file."
5. Click the green button "Create repository."

GitHub is a cloud storage service. All the projects existing on GitHub right now are in the cloud. So naturally, our first step is to get the projects we want to work on off the cloud and onto our local computer. We do this by "cloning" a repository. The term "cloning" makes sense because we are taking an exact copy of the project from the GitHub cloud and putting it onto our computer. The following code clones the repository you want to work in and saves it to your computer. **Remember:** Before you do this, make sure you are in the working directory were you want to save this repository.

After you've created your remote repository following the steps above, GitHub will show you a page with a blue box at the top that says **Quick setup**. Copy the link you see. Now, go back to your terminal and type `git clone` and paste the link you copied and hit enter. The following example shows what you should see in your terminal.

```
~ % git clone https://github.com/MCRoche/Trying_Out_Git.git Instead of MCRoche, it should say your
    GitHub username
Cloning into 'Trying_Out_Git'...
```

```
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
```

If you ever find a repository you want to clone to your machine, simply go to that repository on GitHub (here is my `Trying_Out_Git` repository), and click the green button that says "Code." This will show you the link you can copy and paste into your terminal to clone that repository to your local machine.

Now, when we look at what files are in our current working directory, we should see a folder (a.k.a. a repository) called `Trying_Out_Git`.

```
~ % ls  Checking to see what's in your current working directory
Trying_Out_Git  It's the repository you cloned!
```

Now, everything that is part of the repository `Trying_Out_Git` on the GitHub cloud is on your local computer. These are the steps to clone any GitHub repository you have access to onto your own computer. We can verify this by looking inside the new directory and comparing that to what we see online.

```
~ % ls Trying_Out_Git  Checking out what's in the cloned repository
README.md
```

Succinctly, in order to clone a repository: (1) you need to copy the link to that repository, (2) type "git clone" in your terminal[4], (3) paste the copied link, and (4) hit enter. Boom. You've got the repository.

To work in that repository, change your directory to make it your current working directory.

Step 2. Syncing the Repository with RStudio Now it's time to look through that repository you just cloned in RStudio. Open RStudio as you normally would. Now, instead of starting a new file, go to *File > New Project...* A dialog box will open up. It should look like Figure 1a.

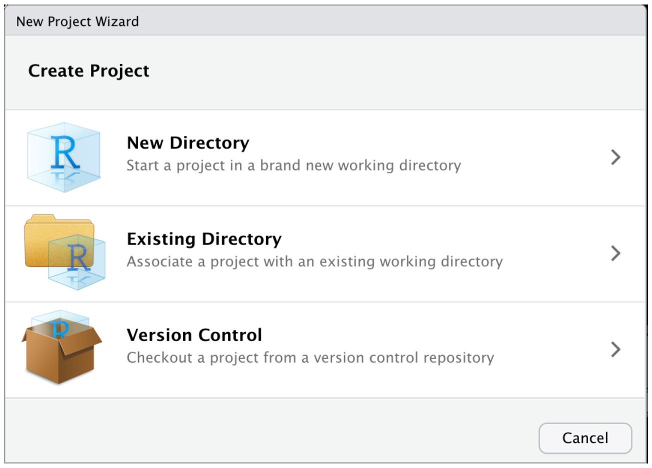
Click on "Version Control." The next window will present the version control options, as seen in Figure 1b. We'll be using git. Click on git.

Now we arrive at the step where we connect our repository to RStudio. The dialog box should look like Figure 1c. In the box for "Repository URL:", paste the same link you copied when you cloned your repository (e.g., `https://github.com/MCRoche/Trying_Out_Git.git`). Name the project directory the same name as your repository (e.g., `Trying_Out_Git`). Finally, decide where you want this project to be saved. I have a folder called "/R_Work", but you can save it anywhere you like (as long as you remember where that is!).

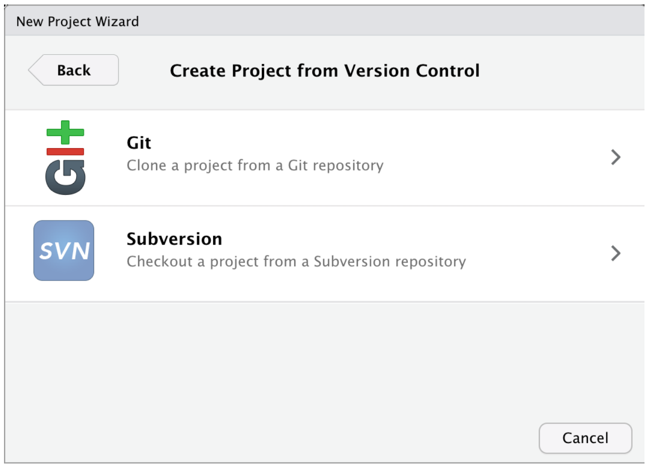
[4]Making sure you are in the working directory you want to be in!

Figure 1: The steps to created a project tracked in git in RStudio.

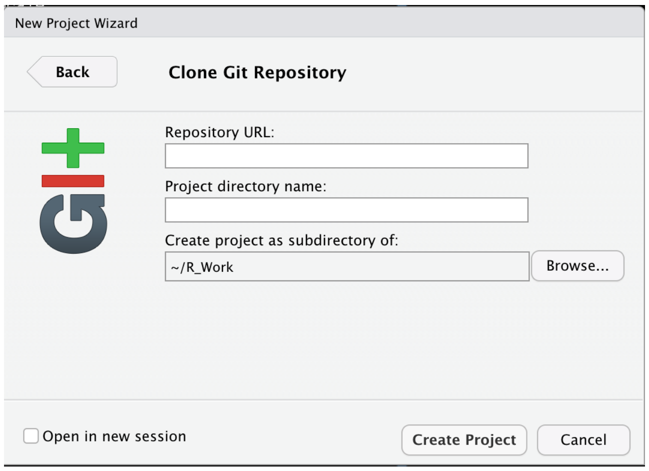
(a) GUI to start a new project in RStudio. This should appear after you click *File > New Project...*


New Project Wizard
Create Project
New Directory
Start a project in a brand new working directory
Existing Directory
Associate a project with an existing working directory
Version Control
Checkout a project from a version control repository
Cancel

(b) GUI to choose which type of version control we want to use. We want git.


New Project Wizard
Back
Create Project from Version Control
Git
Clone a project from a Git repository
Subversion
Checkout a project from a Subversion repository
Cancel

(c) GUI to connect our remote repository to RStudio.



Repository URL:

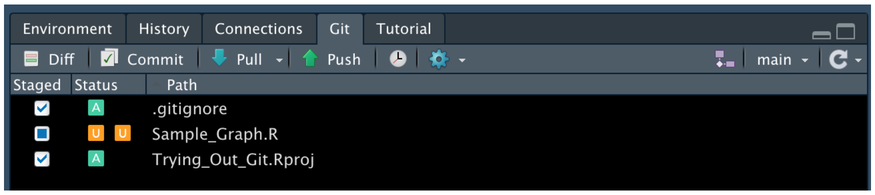
Project directory name:

Create project as subdirectory of: ~/R_Work

Our RStudio project is now all connected to our remote repository! You'll notice there are a few differences in the RStudio setup. If you look over to where environment variables are kept, you'll see a new tab called "Git" (Figure 2a). This provides some nice GUIs and visuals to keep track of what's going on. We're not going to be using it much in this tutorial. Instead, we'll be focused on the other difference: the terminal tab next to the console (Figure 2b).

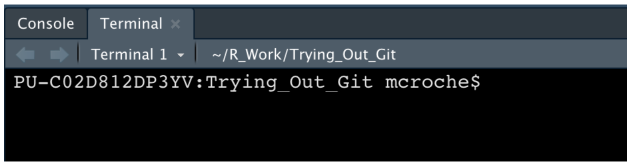
Figure 2: Changes to the RStudio Interface

(a) New Git tab



Staged	Status	Path
✓	A	.gitignore
■	U U	Sample_Graph.R
✓	A	Trying_Out_Git.Rproj

(b) New terminal tab



```
PU-C02D812DP3YV:Trying_Out_Git mcroche$
```

The terminal is the same terminal as the one you were using outside of R. It's another place for you to talk directly to the computer. Now, you don't have to keep switching back and forth between the terminal and R. All the commands we learned in Section 3 will work here. You can see that we're already in our repository and all the files are here. As a check, try seeing what your current working directory is and looking as all the files in it. From now on, we will work from this terminal. The examples will reflect this and you will see a dollar sign $ instead of a tilde and percent sign ~% at the start of a new line.

Step 3. Getting Started Whenever you start working in a repository, it's always a good idea to first check your status. Checking the status tells you differences between the clone of the repository on your computer and the repository on GitHub. To check your status, simple type the following:

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Since we just cloned the repository, there shouldn't be anything different between the cloned repository on our computer, and the repository on GitHub.

Step 4. Creating and Saving You can create a new R file the same as you ever would. Either click the Document with a plus sign in the upper right-hand corner and select "R Script" or hit shift+command+N. To follow along with this tutorial exactly, copy and past the following script into your new R file. Then, save it as Sample_Graph.R. Your current working directory is exactly where you saved your repository when creating this project. You can check this before you save your script by typing “getwd()” in the console.

Checking the current working directory:

```
> getwd()
"/Users/mcroche/R_Work/Trying_Out_Git" #This is were I saved my project. Your path
    will look slightly different
```

R script to copy:

```
##----
##
## Test File: Sample_Graph.R
##
##----

#' This file makes a sample graph.
#' It's point is to demonstrate how to save a file to GitHub and track changes.

library(ggplot2)

# Some points:
x <- data.frame(a=runif(10),
                b=runif(10))

# Some graph:
ggplot(x, aes(a,b)) +
      geom_point() +
      xlab("Hello")
```

Now that you have saved this file as you normally would, we need to save it in git. Currently, the file is saved only at the most local level. We need to tell git that it should pay attention to this file and the changes we make. Whenever we’re using git, we must switch to the terminal instead of the console. Once in the terminal, we add the file to git using **git add**.

```
$ git add Sample_Graph.R
```

If we had multiple files and we wanted to add them all, we could use a period **.**, or dash A **-A** instead of the file name. Now when we check the status again, we will see that we have changes to be committed.

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed: Here git is telling us we have files that need to be committed
      (use "git restore --staged <file>..." to unstage)
            new file: Sample_Graph.R  Here we see which files should be committed
```

The next step, naturally is to commit our changes. Perhaps unsurprisingly, the command to do this is **git commit**. When you commit your files you are saving them to git. ALWAYS INCLUDE A COMMIT MESSAGE. Commit messages are how we communicate what has been changed about the file. To include a commit message, add a dash m **-m** followed by your message in quotes.

```
$ git commit -m "Created a new graph file
[main c1c56ef] Created a new graph file Here's our commit message
 1 file changed, 19 insertions(+) This tells us we added 19 lines of code
 create mode 100644 Sample_Graph.R
```

If you forget to add `−m` to your message, you will be taken to a text editing screen. This screen allows you to write longer, multi-line commit messages. Normally a brief message is fine. If you wind up here and want to get out, first hit the escape key `esc` then type colon, w, q `:wq`. This will take you back to the regular terminal screen. The commits will be aborted due to a lack of commit message, so nothing has changed. Now you can commit your changes with the message as you normally would.

Checking the status once again, we see our working tree is clean!

```
$ git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
      (use "git push" to publish your local commits)

nothing to commit, working tree clean Everything is clean!
```

In the beginning of this tutorial, I mentioned that git and GitHub use three parts: a remote repository, a local repository, and a staging area. When files are successfully committed, they are in the staging area. Our file Sample_Graph.R is committed to our local repository, but it's not yet on the remote repository. The changes are in between the two in the staging area. The importance of the staging area will become clear when we learn how to work as a team. For now, we need our last step to make our file available on our GitHub is to push it from the staging area to the remote repository. The command for this is **git push**.

```
$ git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 521 bytes | 521.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/MCRoche/Trying_Out_Git.git
      3e2f77f..c1c56ef main -> main
```

Now, when you check your online repository, you should see your file!

Putting It All Together Step 4 may seem like a lot, but without my commentary and constantly checking the status (although it's still a good idea to double check the status whenever you want), it's actually pretty quick.

```
First create and locally save a file in R.

$ git add Sample_Graph.R

$ git commit -m "Some message about our file/changes"
[main c1c56ef] Created a new graph file  Here's our commit message
1 file changed, 19 insertions(+)  This tells us we added 19 lines of code
create mode 100644 Sample_Graph.R

$ % git push
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 502 bytes | 502.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://github.com/MCRoche/Trying_Out_Git.git  This is where the file is being sent to
      ce0d04e..88af58b main -> main
```

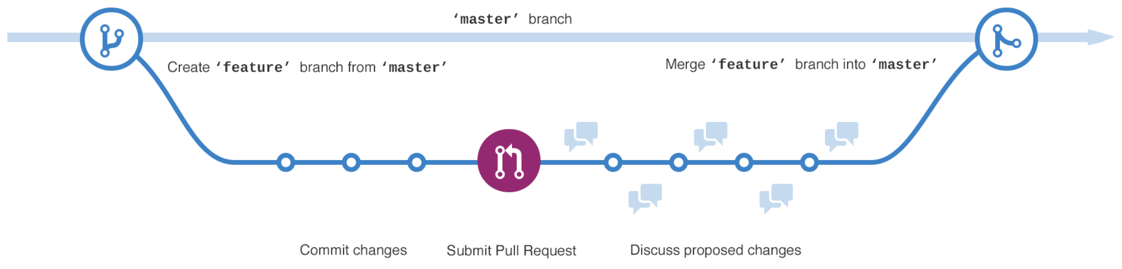
Like everything, the more you do it, the easier it will become.

4.3 Git and GitHub for Team Use

As a review, we learned five git verbs in the previous section: clone, status, add, commit, and push. To successfully navigate our workflow as a team, we need to learn three more: pull, branch, and merge.

So far, git and GitHub seem like neat ways to store files. GitHub's value really shines when you're working with a team where several people are working on the same piece of code. Pulling, branching, and merging is the cornerstone of git. Figure 3 provides a useful visualization of branching and merging. The idea behind branching and merging is to create a space where we can try out new code and have the team review it before adding it to the final version of our project. I'm going to go through each of the commands and how they affect work flow in this section.

Figure 3: Visualization of Branching and Merging in Git


'master' branch
Create 'feature' branch from 'master'
Merge 'feature' branch into 'master'
Commit changes
Submit Pull Request
Discuss proposed changes

Branch You may have noticed in the previous section that whenever we checked our status, the computer began by telling us we were on the main branch. If you didn't notice and don't believe me, you can check the output in the examples I provided. Also, when we have nothing to add or commit, we are told our working tree is clean. The tree and branch metaphor is apt. The main branch (also something called the master branch) is our "trunk" and where we should keep the most finalized project. When we create a branch, we initially create an exact copy of that project. In this branch, we can make changes to code without mucking up our final project. We can see which branches already exist and which branch we are working in with the command **git branch**.

```
$ git branch
* main
```

Currently, there is only the main branch and we are in it. We know this from three things: (1) the branch we are in is starred, (2) the branch we are in is green, and (3) we have not created any other branches so it's impossible to be anywhere else.

Let's create a branch. The command to create a branch is simple. It's just the same command as above followed by what you want to name your branch. Let's call our branch `development` and run **git branch** again to see all our branches and where we are.

```
$ git branch development  Creating a new branch called development
$ git branch
* main
  development
```

Now we can see we have two branch, `main` and `development`, and that we are still in `main`. To switch into our new branch, we have a new command: **git checkout**[5]. After we checkout the development branch, let's run **git branch** again to make sure we're where we want to be.

```
$ git checkout development
Switched to branch 'development'
$ git branch
* development
  main
```

We see that the branch `development` is starred and in green. We're good to go to work in `development`. Right now, `development` is an exact replica of `main`. Any changes we make here, however, will not affect what we've done in `main`. This is the advantage of branching. If we mess up, we can revert back to the code in `main`. If we drastically improve our code, we can merge the branch back into `main` to make the changes permanent.

When we use branching in git, we're not limited to only branching from the `main` branch. We can branch any branch we want, including the ones we just created. Let's branch `development` and call our new

[5]I know; I said only three more verbs. Surprise!

branch `x-label`. Once we created our new branch, we can look at our list of branches again and switch into our new branch.

```
$ git branch x-label| (*Creating a branch of development*)
$ *|git branch
* development
  main
  x-label
$ git checkout x-label
Switched to branch 'x-label'
$ git branch
  development
  main
* x-label
```

Why would we want to branch a branch? Why not just make our changes and merge right back into main? We want to keep main as clean as possible. Making a development branch, and then more branches related to exactly what we're changing allows us to have discussions about what we want to change and what we want to keep. Ideally, we would only merge branches back into main when we are 100% done making changes related to that branch.

Why would I name a branch something weird like `x-label`? When we create new branches, it's useful to name them something related to what we're going to change in the code. This gives our colleagues an idea of what we're working on. Communication is key. If our colleagues see a new branch called `x-label`, they'll know that they don't have to change the x-axis label and to generally avoid making coding changes around there.

Let's make some changes in this branch to get a better idea of what I'm talking about.

First, let's take a look at all the branches, including the once we didn't make.

```
$ git branch -a The –a flag tells us all the local AND remote branches
  development
  main
* x-label
```

The only branches we have right now are the ones we created because, at this point, our colleagues are fictional. Now for the changes. Open your Sample_Graph.R file in RStudio. Currently we have the x-axis labeled as "Hello." Let's change that to "Goodbye."

```
##----
##
## Test File: Sample_Graph.R
##
##----

#' This file makes a sample graph.
#' It's point is to demonstrate how to save a file to GitHub and track changes.
```

```

library(ggplot2)

# Some points:
x <- data.frame(a=runif(10),
                b=runif(10))

# Some graph:
ggplot(x, aes(a,b)) +
      geom_point() +
      xlab("Goodbye")
```

Now, let's save and commit our changes in git. This is the exact same process we went over in Section 4.2. After saving the file locally, we must commit it. Since the file already exists, we can skip the **git add** command and simply use **git commit -am** instead. The **-am** flag tells git to add the file (a) and write a message (m). You can only do this when the file you are changing already exists. You cannot add a new file to your repository this way. DO NOT FORGET THE COMMIT MESSAGE.

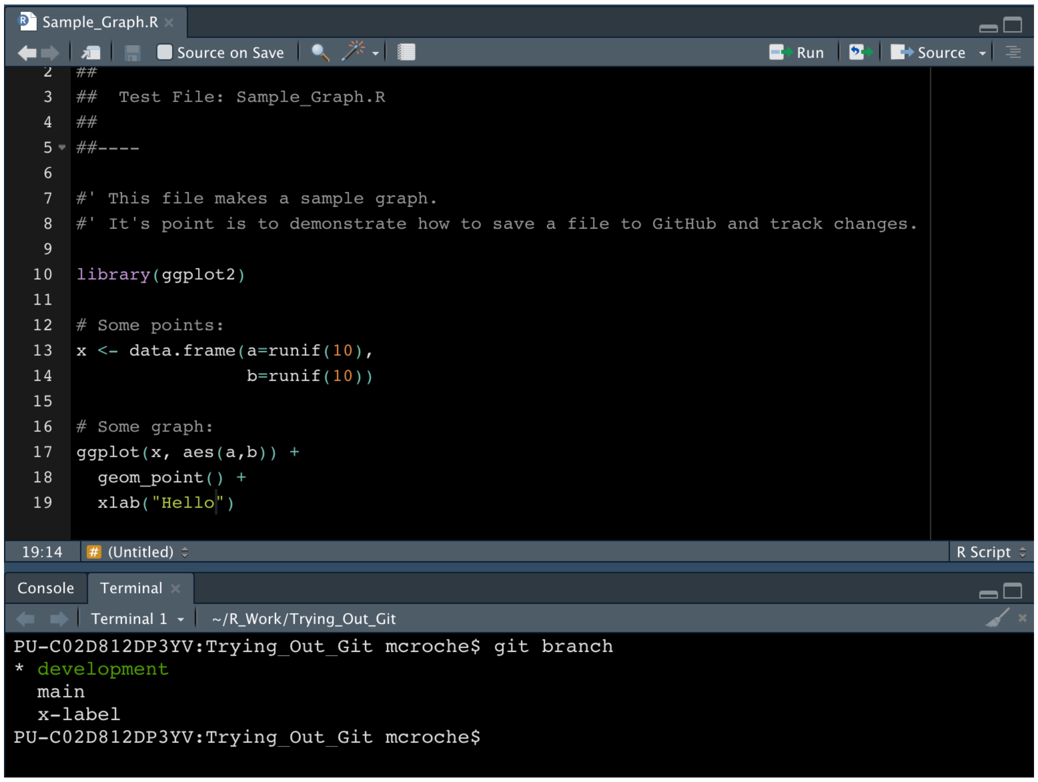
```
$ git commit -am "Changed the x-label Adding and committing changes in one step. Include a commit message
[x-label 9f9ff1a] Changed x-label Here's our commit message
1 file changed, 1 insertion(+), 1 deletion(-) Here's a summary of what we did
```

You can see that git provides a summary of changes. Specifically, it tells us how many lines of code were added and deleted. In this case, we deleted one line ("Hello") and added one line ("Goodbye").

Our change has only been made in the `x-label`. If we switch to the development branch, we see that the label for the x-axis is still "Hello." The comparisons on your screen should look like those in Figures 4a and 4b.

Figure 4: Differences between branches.

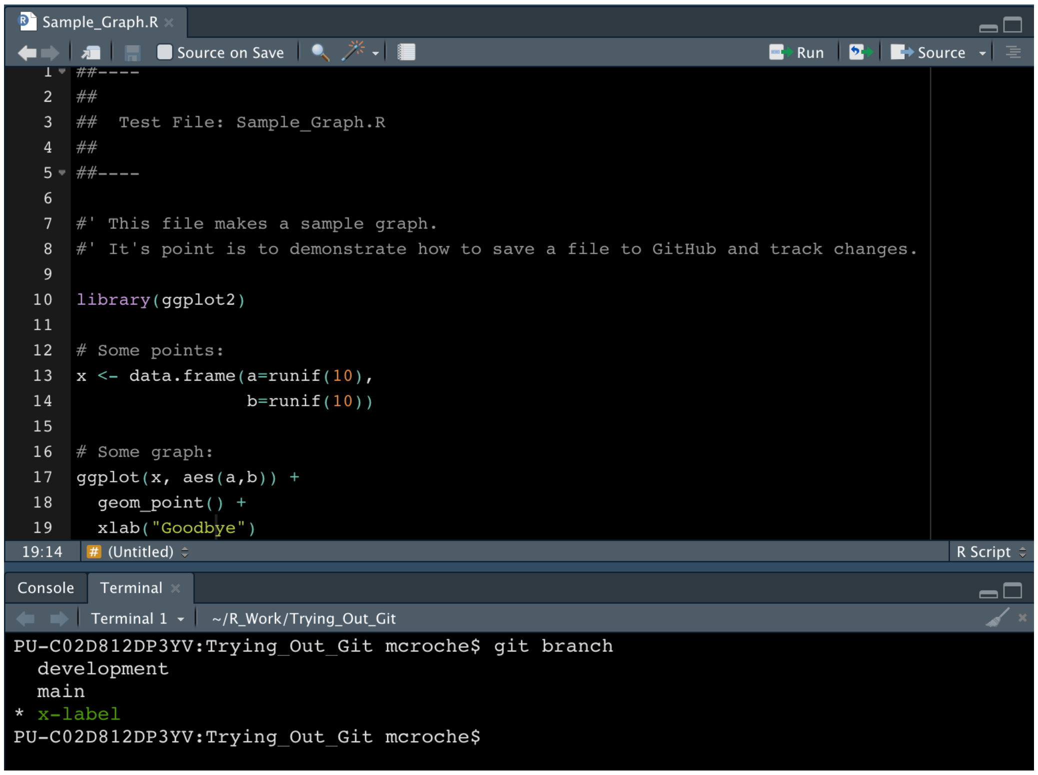
(a) development branch



```
##
##  Test File: Sample_Graph.R
##
##----

#' This file makes a sample graph.
#' It's point is to demonstrate how to save a file to GitHub and track changes.

library(ggplot2)

# Some points:
x <- data.frame(a=runif(10),
                b=runif(10))

# Some graph:
ggplot(x, aes(a,b)) +
  geom_point() +
  xlab("Hello")
```

```
PU-C02D812DP3YV:Trying_Out_Git mcroche$ git branch
* development
  main
  x-label
PU-C02D812DP3YV:Trying_Out_Git mcroche$
```

(b) x-label branch



```
##----
##
##  Test File: Sample_Graph.R
##
##----

#' This file makes a sample graph.
#' It's point is to demonstrate how to save a file to GitHub and track changes.

library(ggplot2)

# Some points:
x <- data.frame(a=runif(10),
                b=runif(10))

# Some graph:
ggplot(x, aes(a,b)) +
  geom_point() +
  xlab("Goodbye")
```

```
PU-C02D812DP3YV:Trying_Out_Git mcroche$ git branch
  development
  main
* x-label
PU-C02D812DP3YV:Trying_Out_Git mcroche$
```

When we used git all by ourselves, this it the point at which we would push our changes to the remote repository. Now we're working in a team though. Before we push, we must...

Pull ALWAYS PULL BEFORE YOU PUSH.

There could be several people working on the same branch. You want to make sure your code is as up to date as possible before you push your changes. A pull request grabs the latest version of the code in your branch. As an example, say someone else was also working on the x-label branch and they added a y-label to the graph. Your code doesn't have that change yet because it wasn't there when you originally branched the project. Before you push your changes, you can make a pull request and the new code your colleague added will be added to your code. To make a pull request, you use the command **git pull**.

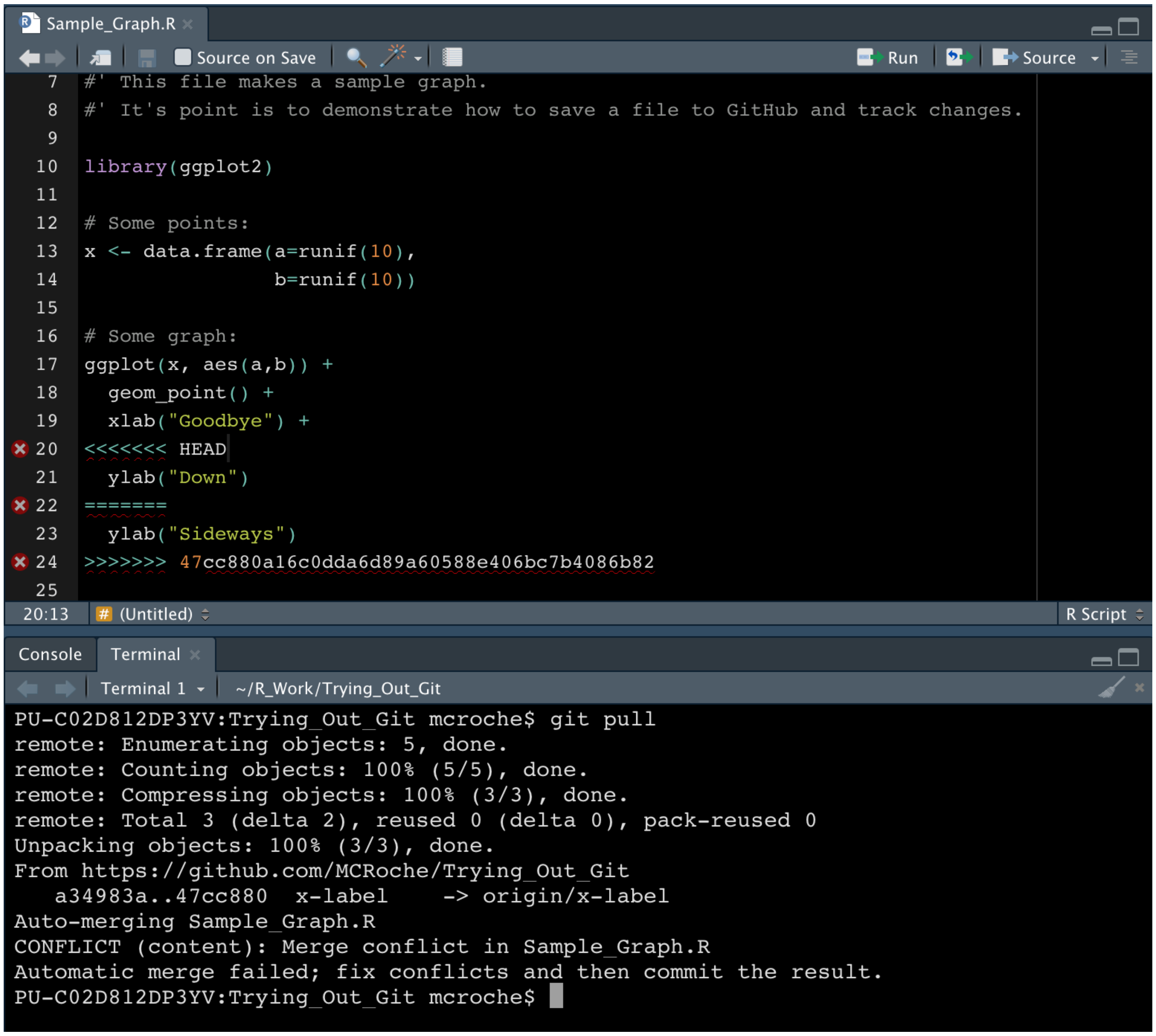
```
I went ahead online and made the y–label change there.
$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/MCRoche/Trying_Out_Git
      a0dca01..a34983a x-label -> origin/x-label
Updating a0dca01..a34983a
Fast-forward
      Sample_Graph.R | 3 ++-
      1 file changed, 2 insertions(+), 1 deletion(-) Now we have 2 insertions because the y–label was
          added in addition to when we wrote "Goodbye"
```

Great! Our code was updated with the latest additions from our colleagues and we didn't have to do much at all. But what we changed the same thing? Say as a team we decided we wanted to change the y-axis label, but we didn't specify what it should be or who should do it. We change the label to "Down" and commit the change. Before we push to the remote repository, we make a pull request, because we remember WE ALWAYS PULL BEFORE WE PUSH. Uh-oh. We see our colleague has also changed the y-axis label and committed it. We see something like the following:

```
$ git pull
remote: Enumerating objects: 5, done
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/MCRoche/Trying_Out_Git
      a34983a..47cc880 x-label -> origin/x-label
Auto-merging Sample_Graph.R
CONFLICT (content): Merge conflict in Sample_Graph.R
Automatic merge failed; fix conflicts and then commit the result.
```

Our RStudio interface looks something like Figure 5. Git is telling us the same line was changed in two places and it doesn't know what to do. We need to correct the conflict manually. We call up our colleague and decide on a name. Then we can push agreed upon version.

Figure 5: Visualization of Branching and Merging in Git



```
#' This file makes a sample graph.
#' It's point is to demonstrate how to save a file to GitHub and track changes.

library(ggplot2)

# Some points:
x <- data.frame(a=runif(10),
                b=runif(10))

# Some graph:
ggplot(x, aes(a,b)) +
  geom_point() +
  xlab("Goodbye") +
<<<<<<< HEAD
  ylab("Down")
=======
  ylab("Sideways")
>>>>>>> 47cc880a16c0dda6d89a60588e406bc7b4086b82

```

```
PU-C02D812DP3YV:Trying_Out_Git mcroche$ git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/MCRoche/Trying_Out_Git
   a34983a..47cc880  x-label    -> origin/x-label
Auto-merging Sample_Graph.R
CONFLICT (content): Merge conflict in Sample_Graph.R
Automatic merge failed; fix conflicts and then commit the result.
PU-C02D812DP3YV:Trying_Out_Git mcroche$
```

Merge conflicts are inevitable. The key to reducing the number of merge conflicts is good communication. The best practice is to commit and pull continuously throughout your workday. This helps you keep on top of all the little changes, so you don't have to go through everything at the end of the day and you can reduce the chance of merge conflicts.

Note: Both you and your colleague who changed the same line have to change your respective versions. If only one of you makes the agreed upon change, you'll still have a pull conflict. If you forget to do a pull request and attempt to push your changes, the computer will yell at you again because it has no idea what to do. It'd be like if you and your friend went out to eat and the wait dropped off one check. You two both put down your credit cards and argue over who will pay. You finally agree that you will cover the tab this time. The waiter comes back to pick up the check, but still sees two credit cards because your friend did not take their credit card away. Now the waiter is mad at both of you.

Speaking of merging...

Merge Merging is the our last step for the code. Now that we all agree on the code in our `x-label` branch, we can merge this branch with the `development` branch. So how do we officially merge our branches `development` and `x-label`? The first step is identify which branch is being merged into which. Here, we want to merge `x-label` in `development`. Ok, let's get back into the `development` branch.

```
$ git checkout development
Switched to branch 'development'
```

To merge the branches, we use the command **git merge**.

```
$ git merge x-label
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 315 bytes | 315.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/MCRoche/Trying_Out_Git.git
      9ea71b0..3bb120c x-label -> x-label
PU-C02D812DP3YV:Trying_Out_Git mcroche$ git checkout development
Switched to branch 'development'
Your branch is ahead of 'origin/development' by 2 commits.
      (use "git push" to publish your local commits)
PU-C02D812DP3YV:Trying_Out_Git mcroche$ git merge x-label
Merge made by the 'recursive' strategy.
      Sample_Graph.R | 3 ++-
      1 file changed, 2 insertions(+), 1 deletion(-)
```

You might be taken to a screen that like the one we saw when we forgot to add the flag **-m** to our commit message. This is the computer demanding to know why you're merging things. It's not a bad idea to tell it. This will also allow anyone in the future to know why you merged branches here. Also, the computer will abort the commit if you don't leave a message. To write a message, hit "i" and then type your message. When you're finished, hit escape `esc` then colon, w, q, `:wq`.

Hooray! You have successfully merged your branches! The `development` branch now looks just like the `x-label` branch. There's only one thing left to do. We now have two copies of the exact same code. We don't need that cluttering up our GitHub. Let's delete the extra branch. Plants grow best when you cut off the dead parts.

There are two things we need to do to delete a branch: (1) delete the branch locally, and (2) delete the branch remotely. Everyone should delete their own local branches when they are done with them. Only one person needs to delete the remote branch.

```
$ git branch -d x-label Deleting a branch locally
Deleted branch x-label (was 3bb120c).
$ git push origin --delete x-label Deleting a branch remotely
To https://github.com/MCRoche/Trying_Out_Git.git
- [deleted] x-label
```

Beautiful. You've survived and are ready to master git. Remember, practice makes perfect.

5 Reviewing Changes

Surprise again! We're not exactly done. An incredibly important part of version control is the ability to review the changes we've made. There are several ways to do this.

5.1 In the Terminal

To look generally at all the changes made to a project, use the command **git log**.

```
$ git log
The latest commit shows up first
commit 082822dc099462cb1d11c8556c9dffcccdf96058 (HEAD -> development) This is the commit hash
Merge: c61b9ac 3bb120c
Author: MCRoche <maryclare.roche@gmail.com> This is who made the commit
Date: Thu Oct 15 18:35:22 2020 -0500 This is when the commit was made

      Merge branch 'x-label' into development This is our commit message

All the commits are listed in reverse chronological order
Skipping to the very first commit

commit ce0d04e146cb1cc3956ee738d06afbec956c9096
Author: MaryClare Roche <maryclare.roche@gmail.com>
Date: Mon Oct 12 11:56:52 2020 -0500

      Initial commit Our commit message when we first created the file
```

With **git log** we can see every change we committed with in this project *in this branch*. It will likely be a long list. To go through the commit log one line at a time, hit `enter`. To go through the commit log one *screen* at a time, hit the space bar. When you've reach the end and reviewed all the commits you want to review, hit the escape key `esc` and then type colon, w, q `:wq`.

You probably noticed the long string of letters and numbers after "commit" at the start of each git log. This is the commit's hash. You can review a specific commit by copying and pasting it's hash after the command **git show**.

```
$ git show ce0d04e146cb1cc3956ee738d06afbec956c9096
commit 082822dc099462cb1d11c8556c9dffcccdf96058 (HEAD -> development)
Merge: c61b9ac 3bb120c
Author: MCRoche <maryclare.roche@gmail.com>
Date: Thu Oct 15 18:35:22 2020 -0500

      Merge branch 'x-label' into development
```

This is especially useful when dealing with merge conflicts. Remember in Figure 5 where there was a long line of what looked like gibberish at line 24? That was the commit hash created when the conflicting commit was made. You can copy and paste that into the terminal after the command **git show** to see details about the commit that your colleague made.

5.1.1 Useful Flags

–all Remember, this is just the commits for the branch you are currently on. To view ALL the commits, add the **–all** flag.

```
$ git log --all
Now all the commits ever made will show up
```

-n There's a lot of output to sift through here. The more people on the team and the longer the project goes, the more commit messages there will be to review. If you only want the see the most recent few commits, you can include a dash and the number of commits you want to see. Here, *n* equals the number of recent commits you want. Let's say we want to review the last three commits.

```
$ git log -3
commit 082822dc099462cb1d11c8556c9dffcccdf96058 (HEAD -> development) Latest commit
Merge: c61b9ac 3bb120c
Author: MCRoche <maryclare.roche@gmail.com>
Date: Thu Oct 15 18:35:22 2020 -0500

      Merge branch 'x-label' into development Latest commit message

commit 3bb120cbacabb272129b1ec178d0416ab75dcf19 Penultimate commit
Author: MCRoche <maryclare.roche@gmail.com>
Date: Thu Oct 15 18:34:51 2020 -0500

      Added a title Penultimate commit message

commit c61b9ac10608d45cbfd502cad40a32acd4467ba2 Antepenultimate commit
Merge: 41dc0d6 9ea71b0
Author: MCRoche <maryclare.roche@gmail.com>
Date: Thu Oct 15 18:05:35 2020 -0500

      Merge branch 'x-label' into development Antepenultimate commit message
```

–author <name> What if you only wanted to review the commits make by one person in particular? You can filter through commits by the author! Cool! Instead of < name >, include the name of the person that would show up in the author field.

```
$ git log -1 Looking at the latest 1 commit.
commit 082822dc099462cb1d11c8556c9dffcccdf96058 (HEAD -> development) Latest commit
Merge: c61b9ac 3bb120c
Author: MCRoche <maryclare.roche@gmail.com> <–– Here's the Author Field
Date: Thu Oct 15 18:35:22 2020 -0500

      Merge branch 'x-label' into development Latest commit message

$ git log --author MCRoche
Now all the commits that I made will show up
```

–before/–after <date> Git is so clever we can even look up commits made one week ago, before yesterday, after five days ago, or between three and four months ago. Specify the dates you're investigating with at string in the format "YYYY-MM-DD." Or, if you know Ruby, you can include Ruby expressions. We specify our date ranges with the **–before** and **–after** tags.

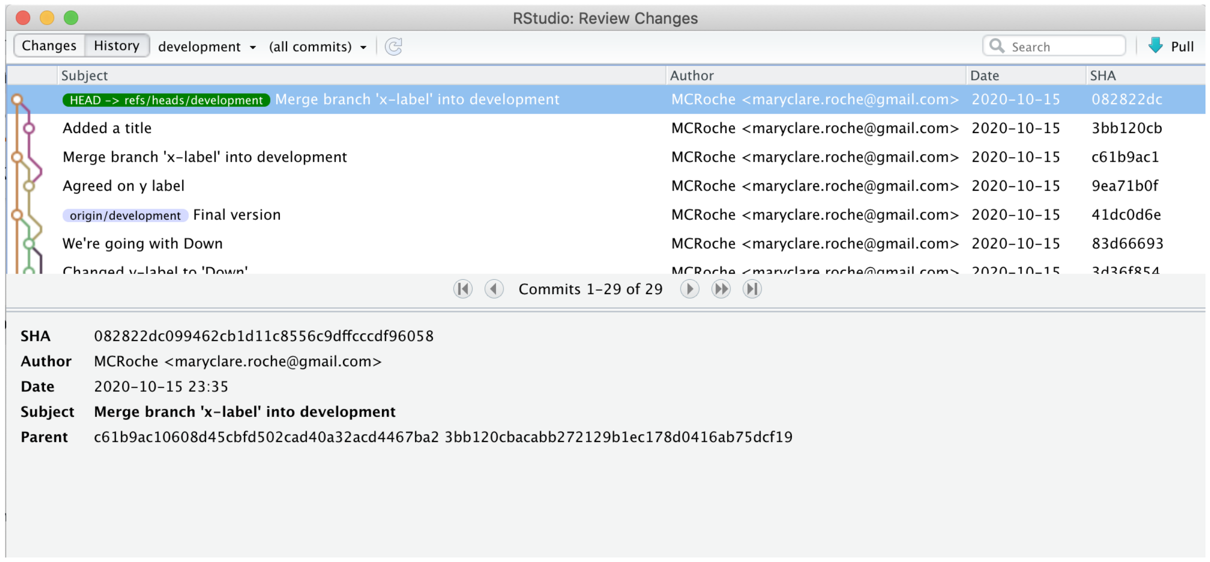
```
$ git log --after 1.day.ago This is Ruby syntax
All the commits you made will show up here since we made all our commits today.
$ git log --after "2020-10-09" --before "2020-10-14" Specifying a range using the string format
Nothing will show up here because we didn't make any commits then.
```

Everything Else This is by no means all you can do to sort through your commits. As I mentioned, git is very clever. These are simply what I think are the most useful when we're all starting out. I encourage to explore the internet for more ideas.

5.2 In RStudio

Ok, ok, I've tortured your enough with the command line. Now I'll let you in on some GUI secrets. Remember that Git tab from Figure 2a? We can click on things over there. Specifically, to review our commits, we can click on the little clock button. This will take us to a screen that looks like Figure 6. On the very left in the top box is a visual map of our changes. Moving to the right, we see the commit messages we added. Next to that, we have the author of the changes, then the date, then the SHA code. We can click on any commit, and it's details will show up in the box underneath

Figure 6: Commit history RStudio GUI.



Subject	Author	Date	SHA
HEAD -> refs/heads/development Merge branch 'x–label' into development	MCRoche <maryclare.roche@gmail.com>	2020–10–15	082822dc
Added a title	MCRoche <maryclare.roche@gmail.com>	2020–10–15	3bb120cb
Merge branch 'x–label' into development	MCRoche <maryclare.roche@gmail.com>	2020–10–15	c61b9ac1
Agreed on y label	MCRoche <maryclare.roche@gmail.com>	2020–10–15	9ea71b0f
origin/development Final version	MCRoche <maryclare.roche@gmail.com>	2020–10–15	41dc0d6e
We're going with Down	MCRoche <maryclare.roche@gmail.com>	2020–10–15	83d66693
Changed y–label to 'Down'	MCRoche <maryclare.roche@gmail.com>	2020–10–15	3d36f854

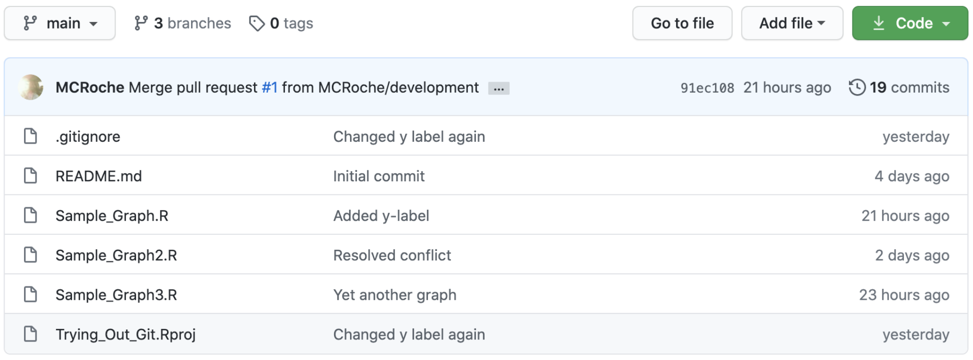
SHA	082822dc099462cb1d11c8556c9dffcccdf96058
Author	MCRoche <maryclare.roche@gmail.com>
Date	2020–10–15 23:35
Subject	**Merge branch 'x–label' into development**
Parent	c61b9ac10608d45cbfd502cad40a32acd4467ba2 3bb120cbacabb272129b1ec178d0416ab75dcf19

5.3 Online

If using the RStudio GUI isn't enough clicking for you, we can also go to the repository's webpage online. You'll see a box like the one in Figure 7a. You can click on the little rewind clock on the right that tells you how many commits have been made in this project. After you click on the rewind clock, you'll be taken to a page with a history of all your commits (See Figure 7b). You can click on any commit to get more details about it. You'll be able to see exactly which lines of code were changed to what (See Figure 7c). Furthermore, you can comment on the commit and the author of the commit can reply.

Figure 7: Stages of Reviewing Commits Online

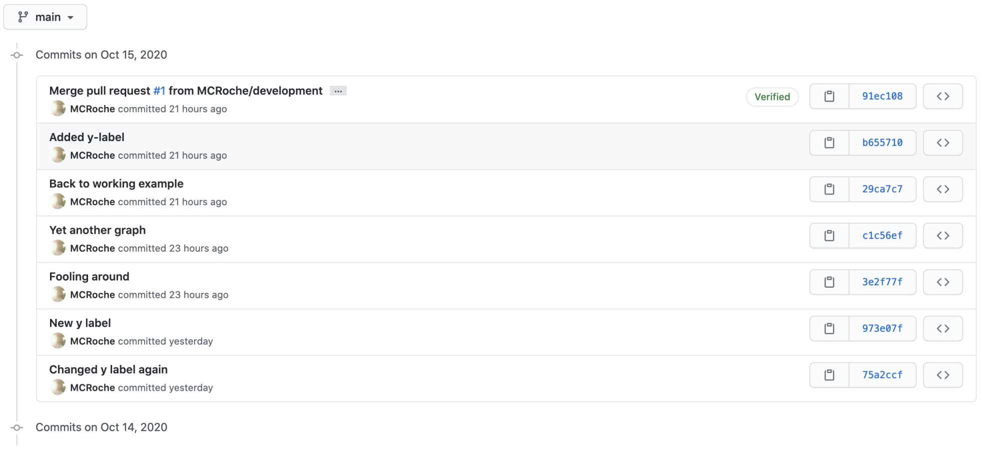
(a) Online Git Repository Webpage


main · 3 branches · 0 tags · Go to file · Add file · Code

MCRoche Merge pull request #1 from MCRoche/development … 91ec108 21 hours ago 19 commits

.gitignore	Changed y label again	yesterday
README.md	Initial commit	4 days ago
Sample_Graph.R	Added y-label	21 hours ago
Sample_Graph2.R	Resolved conflict	2 days ago
Sample_Graph3.R	Yet another graph	23 hours ago
Trying_Out_Git.Rproj	Changed y label again	yesterday

(b) History of Commits Online

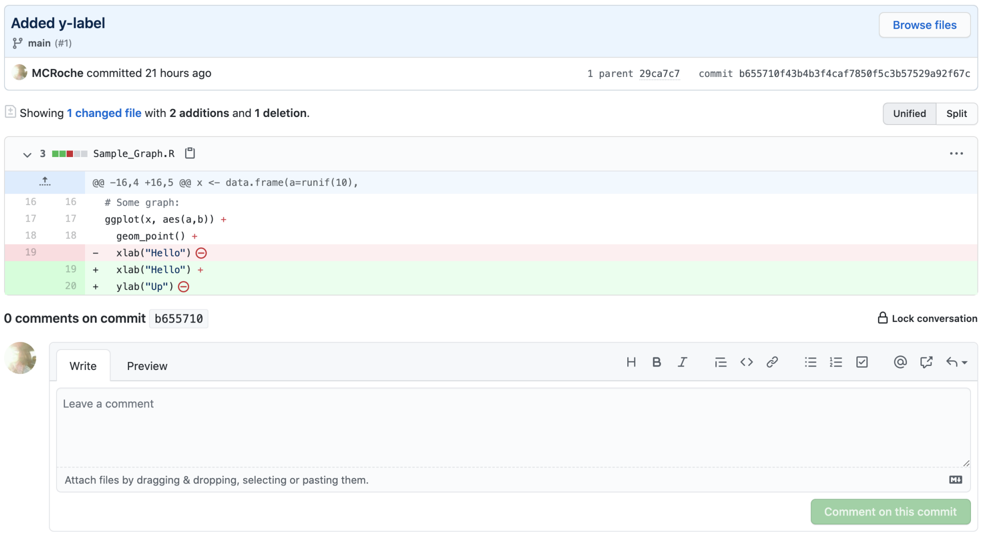

main

Commits on Oct 15, 2020

Merge pull request #1 from MCRoche/development … MCRoche committed 21 hours ago	Verified	91ec108
Added y-label MCRoche committed 21 hours ago		b655710
Back to working example MCRoche committed 21 hours ago		29ca7c7
Yet another graph MCRoche committed 23 hours ago		c1c56ef
Fooling around MCRoche committed 23 hours ago		3e2f77f
New y label MCRoche committed yesterday		973e07f
Changed y label again MCRoche committed yesterday		75a2ccf

Commits on Oct 14, 2020

(c) Details of Commits Online


Added y-label · Browse files

main (#1)

MCRoche committed 21 hours ago · 1 parent 29ca7c7 · commit b655710f43b4b3f4caf7850f5c3b57529a92f67c

Showing 1 changed file with 2 additions and 1 deletion. · Unified · Split

3 Sample_Graph.R

```
@@ -16,4 +16,5 @@ x <- data.frame(a=runif(10),
   # Some graph:
   ggplot(x, aes(a,b)) +
     geom_point() +
-    xlab("Hello")
+    xlab("Hello") +
+    ylab("Up")
```

0 comments on commit b655710 · Lock conversation

Write · Preview

Leave a comment

Attach files by dragging & dropping, selecting or pasting them.

Comment on this commit

6 The End

Congratulations! Now you have *actually* survived! The amount of information in this tutorial may feel overwhelming right now, but the more you use git, the more it will become second nature. There are SO MANY other resources out there about git and GitHub generally. Even though this tutorial was RStudio specific, anything you read elsewhere about git is still applicable here. I recommend checking out YouTube videos, because I find it helpful to watch someone go through what they are explaining to me. Here's a list of some online tutorials and videos about git that I have found helpful.

Useful Vidoes :

- Github Tutorial for Beginners by LearnCode.academy. This is the first video I direct people to when they want to learn git. I find the explanations here super straightforward and intuitive. The video doesn't go over everything in this guide, but it's enough to get your started. The demonstrations regarding merge conflicts and pull requests are great. This video goes over most topics in Section 4.2.
- Github Pull Request, Branching, Merging & Team Workflow by LearnCode.academy. This video picks up where the last one left off. It goes over what's covered in Section 4.3. The narrator is the same, so his explanations are just as great as they were in the first video.
- Corey Schafer's Git Tutorial Playlist. There are six videos in this tutorial. The fist video goes over the basics while the rest are good to watch once you become more comfortable with git. He writes all his examples using Sublime text[6] and .html files. Don't let these differences deter you; everything he does will work the same way in the RStudio terminal. In general, Corey Schafer's YouTube channel is a great place to start when you want to learn how to do computer stuff.

Useful Webpages :

- GitHub Docs homepage. This is where all the GitHub documentation can be found. There's so much more than what I covered here.
- git homepage. Remember from the very beginning when I clarified that git and GitHub are two different things? Well, here's git's homepage with all its documentation. On its documentation page, you can find a reference manual, a book, and videos all designed to help you use git.
- When nothing here makes sense, this website always has the answers.

[6]Sublime text is a pretty cool, free text editor.

Figure 8: Congratulation. You're now...

(a) Picture from Mr. Robot