Intro to Git and GitHub, Season 1

October, 2023

Table of Contents

# 1 About this Course

You will learn how to use Git, a version control system that is the primary means of doing reproducible and collaborative research. You will use Git from the command line to document the history of your code, create different versions of your code, and share your code with an audience via GitHub!

Prerequisites: Completion of Intro to Command Line or demonstrating competency.

Targeted Audience: Researchers who want to keep track the history of their code at a professional standard, and share it with an audience.

# 2 Introduction to Git and GitHub

## 2.1 Why version control?

As the book [ProGit](https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control) puts it, “Version control is a system that records changes to a file or set of files over time so that you can recall specific versions later”. You have tried some form of version control yourself: the simplest form of version control consists of carefully duplicating and organizing your files that you have a history of your edits, but that is prone to inconsistency and errors.

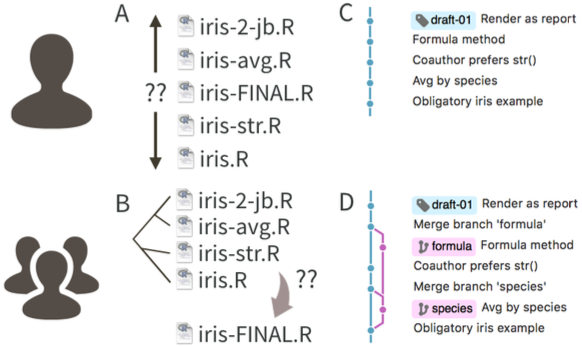


Figure : A: Solo work with DIY version control via filename. B: Collaborative work with DIY version control. C: Solo work with Version Control. D: Collaborative work with Version Control.

People developed simple version control software that keeps tracks the version of your files in a database, which contains metadata of timestamp, author, and version notes.

That’s great, but if there are multiple people working on the project, then a server should host the project’s history and let users create new features and resolve conflicts of the project. **Git** was developed as a distributed version of this collaborative concept, meaning that each user has a complete history of the project. So if the server goes down, each user still has a copy of the project’s history.

**GitHub** is a website that allows you to store Git projects remotely online. It facilitates collaboration, open-source coding, and provides user-friendly features to work on multiple versions of a project simultaneously.

In this workshop we will focus on the fundamentals of Git, and use GitHub only to promote collaboration and sharing.

## 2.2 The Git Data Model

Let’s formalize Git’s version control system: Git keeps track of a project within a designated directory, which is called a **repository** (also known as **repo**). After you make some changes to the files in your repository, you can save the state of your repository by asking Git to make a **commit**. When making a commit, Git will save the repository’s **directory tree** as well as a link to the **parent commit**.

To reduce redundancy in storing the commit information, Git will see what has changed in the repository relative to the previous version. If a file in the repository has not changed at the time of the commit, Git doesn’t store the file again, but instead create a link to the previous file as it was already stored. (diff as an example)

Below is an illustration of a repository’s commit history. Each version (commit) points to a parent version (commit), and a directory tree is associated with each commit. Some, but not all, files are changed at each commit.



Figure : Git model with a linear history. (Source: ProGit)

What is a reasonable amount of changes you should have before making a commit? Well, every time you make a commit, you have to submit a concise **commit message**. This encourages good documentation of the changes you made, and usually you want to make a commit when you have made a modular change to the repository you have been working on.

Another way of illustrating commits: each o represents a commit (consisting of an entire directory tree), and points to a parental commit (Source: [MIT’s Git Seminar](https://missing.csail.mit.edu/2020/version-control/))

o <-- o <-- o <-- o

It is possible to create a branch structure, in a situation where two separate features of the project are being made in parallel:

o <-- o <-- o <-- o  
 ^  
 \  
 --- o <-- o

When both features are complete, one could merge them together to have an unified, unbranched repository:

o <-- o <-- o <-- o <---- o  
 ^ /  
 \ v  
 --- o <-- o

This workflow of branching and merging is extremely popular in collaborative work, and we will hold future seminars on mastering it.

## 2.3 Set up

### 2.3.1 Installing Git

If you are learning via replit, then Git is already installed. See this guide for [installation info](https://git-scm.com/book/en/v2/Getting-Started-Installing-Git) on your local computer.

### 2.3.2 Create a GitHub account

Create your GitHub account [here](https://github.com/login).

### 2.3.3 Adding your information for Git

This helps Git record your author information when you make commits:

% git config --global user.name "Your Name"  
% git config --global user.email "address@email.com"

Replace Your Name with your name and address@email.com with your email address you used for your GitHub account.

### 2.3.4 Connecting Git to your GitHub account

We change the settings of your Git software so that it connects to your GitHub account:

% gh auth login

You will be asked how you want to log in, and pick the following:

? What account do you want to log into? GitHub.com  
? What is your preferred protocol for Git operations? HTTPS  
? Authenticate Git with your GitHub credentials? Yes  
? How would you like to authenticate GitHub CLI? Login with a web browser

You will be given a code, and you will provide that code to GitHub via <https://github.com/login/device>.

### 2.3.5 Optional: change text editor

We will tell Git what text editor to use to create commit messages. If you are new to command line text editors, nano is a simple one to learn. Other common ones are vim and emacs.

% git config --global core.editor "nano"

## 2.4 Setting up a local repository

We create a local repository by using the command git init [repository name]

% mkdir sandbox  
cd sandbox  
git init  
Initialized empty Git repository.

Git has created a hidden folder .git/ that will store all the commit version history. We will take a quick look, but the it is Git’s job to take care of this folder.

% ls -a  
. .. .git  
% cd .git   
% ls -l   
total 24  
-rw-r--r-- 1 clo2 staff 21 Sep 5 16:15 HEAD  
-rw-r--r-- 1 clo2 staff 137 Sep 5 16:15 config  
-rw-r--r-- 1 clo2 staff 73 Sep 5 16:15 description  
drwxr-xr-x 15 clo2 staff 480 Sep 5 16:15 hooks  
drwxr-xr-x 3 clo2 staff 96 Sep 5 16:15 info  
drwxr-xr-x 4 clo2 staff 128 Sep 5 16:15 objects  
drwxr-xr-x 4 clo2 staff 128 Sep 5 16:15 refs

Let’s exit this folder and run git status to understand the current state of this repository.

% cd ..  
% git status  
On branch main  
  
No commits yet  
  
nothing to commit (create/copy files and use "git add" to track)

We will create some files in this repository, and make our first commit. Notice that in the message of git status, it gives suggestions on what you can do. We will create a file, and commit it to our repository. Before we do so, we have some more concepts to unpack.

## 2.5 Staging Model to make commits

In addition to the Git Data Model, there is a Staging Model to keep in mind when trying to make commits. In the simplest version control system, one could imagine a commit command that takes a snapshot of the repository’s directory and stores it as the commit. In Git, there is a intermediate **staged** state so that you can selective decide which files from your repository should be in the commit.

Why offer this intermediate staging ground? Perhaps, in a rush, you implemented two modular changes to your code in two different files, respectively. You want to make two separate commits, one for each change, so you stage one file, commit that one file, and then stage the second file, and then commit that second file. Or perhaps, during your analysis, you generated a bunch of temporary files that isn’t necessary for the software to run, so you don’t stage these temporary files to commit. Or perhaps, *you have sensitive PHI data in your repository’s directory and should not commit it*.

From [ProGit](https://git-scm.com/book/en/v2/Getting-Started-What-is-Git%3F), once Git **tracks** your file, Git has three main states that your files can reside in: **modified**, **staged**, and **committed**:

* **Modified** means that you have changed the file but have not committed it to your local repository yet.
* **Staged** means that you have marked a modified file in its current version to go into your next commit.
* **Committed** means that the data is safely stored in your local repository.

## 2.6 Making your first commit

Create a file named README, and add some text to it using a text editor, such as nano.

% touch README

Let’s look at the status:

% git status  
On branch main  
Untracked files:  
 (use "git add <file>..." to include in what will be committed)  
 README  
  
nothing added to commit but untracked files present (use "git add" to track)

We see that README is **untracked**, which means Git does not consider the file relevant for the repository yet. Let’s add it via git add.

% git add README   
% git status  
On branch main  
Changes to be committed:  
 (use "git restore --staged <file>..." to unstage)  
 new file: README

Now, README is “staged” (and thus also “tracked”). We will commit it to our repository using git commit and option -m to write a short commit message. If you don’t specifiy the -m option, Git will open up your defaul text editor and ask you to write something.

% git commit -m "Added README"  
[main bb926a2] Added README  
 1 file changed, 1 insertion(+)  
 create mode 100644 README  
clo2@MGQQR2YQRT9 sandbox % git status  
On branch main  
nothing to commit, working tree clean

Great, you have completed your first commit! README is “committed”.

Now, let’s modify README again. What’s the status?

% nano README   
% git status  
On branch main  
Changes not staged for commit:  
 (use "git add <file>..." to update what will be committed)  
 (use "git restore <file>..." to discard changes in working directory)  
 modified: README

README is in the “modified” state. Notice that unlike the first time we made changes to README, it is already “tracked”, because it was “committed” in the last commit. If we use git add, then it will be “staged”:

% git add README   
% git status  
On branch main  
Changes to be committed:  
 (use "git restore --staged <file>..." to unstage)  
 modified: README

You should finish the commit by git commit.

## 2.7 Staging model revisited



## 2.8 Exercises

* Make some changes and make another commit by yourself.
* Modify README, stage it via git add, and then modify README again. look at git status. How can README file be staged and unstaged at the same time?
* Moving a file: Rename a committed file with mv. Then, look at git status. What do you need to do to commit this change? Do the necessary steps and commit it. Repeat this exercise again with git mv. This should be easier.
* Deleting a file: delete a committed file with rm. Then, look at git status. What do you need to do to commit this change? Do the necessary steps and commit it. Do this again with git rm. This should be easier.

### 2.8.1 Tricks

* Adding the -u option to git add makes Git automatically stage every file that is already tracked before.
* Adding the -a option to git commit makes Git automatically stage every file that is already tracked before, and also executes the commit. ie. it runs git add on all previously tracked files and commits all in one command.
* If you made a commit but forgot a file, add it via git add, and then git commit --amend to amend your commit to contain your file.

## 2.9 Git log

Let’s take a look of the history of our repository by running git log:

commit 5da6d498e47bb8e5b4294a10a153275bed166004 (HEAD -> main)  
Author: Chris Lo <hidden@gmail.com>  
Date: Wed Sep 6 13:36:48 2023 -0700  
  
 Added author name to README  
  
commit 30872fd1de1ac8998c8dac8419937cd59d9d2282  
Author: Chris Lo <hidden@gmail.com>  
Date: Wed Sep 6 12:15:44 2023 -0700  
  
 Added README

Each commit is associated with an identifier of numbers and letters, for security purposes. We will refer to these commit identifiers if we need to look back into the history of our commits.

Two other things to notice, that we will dive deeper in the future:

The HEAD **pointer** tells us “What am I looking at?” in our commit history. If we want to look back into the history of the commits, then we need to move the HEAD pointer. main refers to the **main branch** of this repository. If we create different branches, then we need to make a distinction between the main and other branches created. In summary, we are looking at the most recent commit on the main branch, because HEAD is pointing to it.

### 2.9.1 Looking back at history

We can move the HEAD pointer to see the history of this repository. Let’s point it to an older commit:

% git checkout 30872fd1de1ac8998c8dac8419937cd59d9d2282  
Note: switching to '30872fd1de1ac8998c8dac8419937cd59d9d2282'.  
  
You are in 'detached HEAD' state. You can look around, make experimental  
changes and commit them, and you can discard any commits you make in this  
state without impacting any branches by switching back to a branch.

Don’t worry about this “detached HEAD” message for now. If you look at README, you will see that you are looking at an older version! If you run git log, you will see that the HEAD is pointing at the commit you specified. We are indeed keeping track of our history. Let’s return back to the latest commit:

% git checkout main   
Previous HEAD position was 23ee460 Merge branch 'main' of https://github.com/caalo/sandbox  
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.

### 2.9.2 Other ways of looking at git log:

* Try git log --all --graph --decorate to see a more decorative illustration.
* Try git log --patch for a verbose output. This gives us a sense how git is keeping track of all the changes in the .git folder.

## 2.10 Undoing

### 2.10.1 Unstage a file

Suppose you staged a file, and then realized you don’t want the file to be staged for the commit. To unstage the file so that it returns to “modified”, run git restore --staged [fileName].

### 2.10.2 Unmodify a file

After your commit, you make some new changes to a file so that it is “modified” You realized that you are no longer interested in this modification and want to un-modify it back to the last commit. Run: git restore [fileName].

*Warning: This cannot be undone!*

### 2.10.3 Revert a commit

After several commits, you realized that you wanted to undo a specific commit. Run: git revert [commit identifier]. This will undo that commit. However, if the content of that commit has experienced intermediate changes, it will raise a conflict.

### 2.10.4 Reset to a previous commit

To “time-travel” back to a previous commit, run: git reset --hard [commit identifier].

*Warning: This cannot be undone!*

## 2.11 Creating a GitHub remote repository

So far, we have talked about how to use your Git repository to keep track of your own work, but a big feature of Git is that you can collaborate with others. To do so, **remotes** are repositories hosted often on a server, such as GitHub, so that other people can access the the remote from their local computer.

We first create a GitHub remote repository on the GitHub account.

## 2.12 Connecting to your GitHub remote repository

Let’s return to our local repository. We want to connect our local repository to the remote repository you just created. Then, once we are connected, we want to **push** our commits to the remote so that other people can access it.

If you run git remote, you should not see any messages, because your local repository currently is not connected to any remote repositories.

We also run git log to see what the status of our repository before we connect to the remote:

% git log   
commit 113a880be8eccf51d514ed4cedee0111e399bfad (HEAD -> main)  
Author: Chris Lo <hidden@gmail.com>  
Date: Thu Sep 7 13:19:41 2023 -0700  
  
 Added author name to README

Now, let’s make the connection: the syntax is git remote add [remote name] [Git remote URL]:

% git remote add origin https://github.com/caalo/sandbox.git

We gave the remote name origin. This is a standard remote name for GitHub that you should use.

Let’s push our commits to the remote:

% git branch -M main  
% git push -u origin main  
Enumerating objects: 3, done.  
Counting objects: 100% (3/3), done.  
Writing objects: 100% (3/3), 218 bytes | 218.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
To https://github.com/caalo/sandbox.git  
 \* [new branch] main -> main

The first line makes sure that the branch we are working with locally is called main, and we are doing it to be safe.

The -u option is needed for our first push to a new repository, making the connection between main and origin - we set origin “upstream” of main. After we make this upstream definition, we can use git push without these options.

Let’s look at our log again:

% git log   
commit 113a880be8eccf51d514ed4cedee0111e399bfad (HEAD -> main, origin/main)  
Author: Chris Lo <hidden@gmail.com>  
Date: Thu Sep 7 13:19:41 2023 -0700  
  
 Added author name to README

Our latest commit is associated with origin/main, which refers to the remote name origin and the main branch on the remote. Everything is synced up between our local repository and the remote repository. You can see your GitHub website updated with the changes.

Now, here is what our staging model looks like:



If you make a new commit, but don’t push it yet, what happens?

% nano README  
% git commit -m "Edited README to show GitHub"  
[main e0ce6cd] blob  
 1 file changed, 1 insertion(+), 1 deletion(-)  
% git log  
commit e0ce6cd822b7b3affd6b6b624520316cc375991f (HEAD -> main)  
Author: Chris Lo <chris.aa.lo@gmail.com>  
Date: Thu Sep 7 14:30:57 2023 -0700  
  
 Edited README to show GitHub  
  
commit 23ee460e4590c87f6676d8085576ef8902d5ea51 (origin/main)  
Merge: 5da6d49 b261e9d  
Author: Chris Lo <chris.aa.lo@gmail.com>  
Date: Thu Sep 7 13:11:33 2023 -0700  
  
 Added author name to README

Our HEAD pointer to our main branch is “ahead” of the remote origin’s main branch! That makes sense because we only made a local commit to our repository. To update our remote,

% git push  
Enumerating objects: 5, done.  
Counting objects: 100% (5/5), done.  
Writing objects: 100% (3/3), 259 bytes | 259.00 KiB/s, done.  
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0  
To https://github.com/caalo/sandbox.git  
 23ee460..e0ce6cd main -> main  
% git log  
commit e0ce6cd822b7b3affd6b6b624520316cc375991f (HEAD -> main, origin/main)  
Author: Chris Lo <chris.aa.lo@gmail.com>  
Date: Thu Sep 7 14:30:57 2023 -0700  
  
 Edited README to show GitHub

Both the local repository’s main branch and remote repository’s main branch are synced.

## 2.13 Future seminars

We will talk more about collaboration using branches and pull requests in the future. If you have active collaborators who interact with GitHub using branches and pull requests, you have the knowledge and setup from this seminar to quickly pick up these skills. See our [guide](https://hutchdatascience.org/dasl-snack-github/making-a-pull-request-beginners.html).

## 2.14 Appendix: Other ways of interacting with Git and GitHub

If interfacing with Git is not your favorite way to do so, there are GUI-based software that allows you to use Git interactively. Some popular ones include:

* [Git Kraken](https://www.gitkraken.com/)
* [GitHub Desktop](https://desktop.github.com/)
* [RStudio with usethis package](https://hutchdatascience.org/Tools_for_Reproducible_Workflows_in_R/using-github-in-a-workflow.html)

**A nice troubleshooting guide**

[DangItGit](https://dangitgit.com/en)

## 2.15 Appendix: References

* [ProGit](https://git-scm.com/book/en/v2/Getting-Started-About-Version-Control): We covered chapters 1-2 in this seminar.
* [MIT’s Git Seminar](https://missing.csail.mit.edu/2020/version-control/): A more computer science explanation of how Git works.
* [Explain Shell](https://explainshell.com/): Access Shell and Git manual and help pages in an easy-to-read way.
* Bryan J. 2017. Excuse me, do you have a moment to talk about version control? PeerJ Preprints 5:e3159v2 <https://doi.org/10.7287/peerj.preprints.3159v2>

# About the Authors

These credits are based on our [course contributors table guidelines](https://www.ottrproject.org/more_features.html#giving-credits-to-contributors).

| Credits | Names |
| --- | --- |
| **Pedagogy** |  |
| Lead Content Instructor(s) | Chris Lo |
| Lecturer | Chris Lo |
| Content Author(s) (include chapter name/link in parentheses if only for specific chapters) - make new line if more than one chapter involved | If any other authors besides lead instructor |
| Content Contributor(s) (include section name/link in parentheses) - make new line if more than one section involved | Wrote less than a chapter |
| Content Editor(s)/Reviewer(s) | Checked your content |
| Content Director(s) | Helped guide the content direction |
| Content Consultants (include chapter name/link in parentheses or word “General”) - make new line if more than one chapter involved | Gave high level advice on content |
| Acknowledgments | Gave small assistance to content but not to the level of consulting |
| **Production** |  |
| Content Publisher(s) | Helped with publishing platform |
| Content Publishing Reviewer(s) | Reviewed overall content and aesthetics on publishing platform |
| **Technical** |  |
| Course Publishing Engineer(s) | Helped with the code for the technical aspects related to the specific course generation |
| Template Publishing Engineers | [Candace Savonen](https://www.cansavvy.com/), [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/) |
| Publishing Maintenance Engineer | [Candace Savonen](https://www.cansavvy.com/) |
| Technical Publishing Stylists | [Carrie Wright](https://carriewright11.github.io/), [Ava Hoffman](https://www.avahoffman.com/), [Candace Savonen](https://www.cansavvy.com/) |
| Package Developers ([ottrpal](https://github.com/jhudsl/ottrpal)) [Candace Savonen](https://www.cansavvy.com/), [John Muschelli](https://johnmuschelli.com/), [Carrie Wright](https://carriewright11.github.io/) |  |
| **Art and Design** |  |
| Illustrator(s) | Created graphics for the course |
| Figure Artist(s) | Created figures/plots for course |
| Videographer(s) | Filmed videos |
| Videography Editor(s) | Edited film |
| Audiographer(s) | Recorded audio |
| Audiography Editor(s) | Edited audio recordings |
| **Funding** |  |
| Funder(s) | Institution/individual who funded course including grant number |
| Funding Staff | Staff members who help with funding |

## ─ Session info ───────────────────────────────────────────────────────────────  
## setting value   
## version R version 4.0.2 (2020-06-22)  
## os Ubuntu 20.04.5 LTS   
## system x86\_64, linux-gnu   
## ui X11   
## language (EN)   
## collate en\_US.UTF-8   
## ctype en\_US.UTF-8   
## tz Etc/UTC   
## date 2023-10-18   
##   
## ─ Packages ───────────────────────────────────────────────────────────────────  
## package \* version date lib source   
## assertthat 0.2.1 2019-03-21 [1] RSPM (R 4.0.5)   
## bookdown 0.24 2023-03-28 [1] Github (rstudio/bookdown@88bc4ea)   
## cachem 1.0.7 2023-02-24 [1] CRAN (R 4.0.2)   
## callr 3.5.0 2020-10-08 [1] RSPM (R 4.0.2)   
## cli 3.6.1 2023-03-23 [1] CRAN (R 4.0.2)   
## crayon 1.3.4 2017-09-16 [1] RSPM (R 4.0.0)   
## desc 1.2.0 2018-05-01 [1] RSPM (R 4.0.3)   
## devtools 2.3.2 2020-09-18 [1] RSPM (R 4.0.3)   
## digest 0.6.25 2020-02-23 [1] RSPM (R 4.0.0)   
## ellipsis 0.3.1 2020-05-15 [1] RSPM (R 4.0.3)   
## evaluate 0.20 2023-01-17 [1] CRAN (R 4.0.2)   
## fastmap 1.1.1 2023-02-24 [1] CRAN (R 4.0.2)   
## fs 1.5.0 2020-07-31 [1] RSPM (R 4.0.3)   
## glue 1.4.2 2020-08-27 [1] RSPM (R 4.0.5)   
## htmltools 0.5.5 2023-03-23 [1] CRAN (R 4.0.2)   
## knitr 1.33 2023-03-28 [1] Github (yihui/knitr@a1052d1)   
## magrittr 2.0.3 2022-03-30 [1] CRAN (R 4.0.2)   
## memoise 2.0.1 2021-11-26 [1] CRAN (R 4.0.2)   
## pkgbuild 1.1.0 2020-07-13 [1] RSPM (R 4.0.2)   
## pkgload 1.1.0 2020-05-29 [1] RSPM (R 4.0.3)   
## prettyunits 1.1.1 2020-01-24 [1] RSPM (R 4.0.3)   
## processx 3.4.4 2020-09-03 [1] RSPM (R 4.0.2)   
## ps 1.4.0 2020-10-07 [1] RSPM (R 4.0.2)   
## R6 2.4.1 2019-11-12 [1] RSPM (R 4.0.0)   
## remotes 2.2.0 2020-07-21 [1] RSPM (R 4.0.3)   
## rlang 1.1.0 2023-03-14 [1] CRAN (R 4.0.2)   
## rmarkdown 2.10 2023-03-28 [1] Github (rstudio/rmarkdown@02d3c25)  
## rprojroot 2.0.3 2022-04-02 [1] CRAN (R 4.0.2)   
## sessioninfo 1.1.1 2018-11-05 [1] RSPM (R 4.0.3)   
## stringi 1.5.3 2020-09-09 [1] RSPM (R 4.0.3)   
## stringr 1.4.0 2019-02-10 [1] RSPM (R 4.0.3)   
## testthat 3.0.1 2023-03-28 [1] Github (R-lib/testthat@e99155a)   
## usethis 1.6.3 2020-09-17 [1] RSPM (R 4.0.2)   
## withr 2.3.0 2020-09-22 [1] RSPM (R 4.0.2)   
## xfun 0.26 2023-03-28 [1] Github (yihui/xfun@74c2a66)   
## yaml 2.2.1 2020-02-01 [1] RSPM (R 4.0.3)   
##   
## [1] /usr/local/lib/R/site-library  
## [2] /usr/local/lib/R/library

# 3 References