Stat 8054 Lecture Notes: Git

Charles J. Geyer

January 17, 2023

License

Version

The version of `git` used to make this document is

```
git --version
```

```
## git version 2.39.0
```

Wikipedia

- Version Control
- Git
- GitHub

The Best Book about Git

Pro Git by Scott Chacon and Ben Straub is available to read online for free. Dead tree versions are available on Amazon.com.

Git Web Site

- http://git-scm.com

GitHub

- http://github.com/
- http://github.com/cjgeyer/
- http://github.umn.edu/
- http://github.umn.edu/geyer/

Version Control

- revision control
- version control
- version control system (VCS)
- source code control
- source code management (SCM)
- content tracking

All the same thing.

It isn't just for source code. It's for any "content".

I use it for

- classes (slides, handouts, homework assignments, tests, solutions),
- papers (versions of the paper, tech reports, data, R scripts for data analysis),
- R packages (the traditional source code control, although there is a lot besides source code in an R package),
- notes (long before they turn into papers, I put them under version control).

Very Old Fashioned Version Control

```
asterLA.7-3-09.tex    asterLA.10-3.tex
asterLA.7-16.tex      asterLA.10-3c.tex
asterLA.7-16c.tex     asterLA10-4.tex
asterLA.7-19.tex      asterLA10-4c.tex
asterLA.7-19c.tex     asterLA10-4d.tex
asterLA.7-19z.tex     asterLA01-12.tex
asterLA8-19.tex       asterLA01-12c.tex
asterLA8-19c.tex      asterLA01-20.tex
asterLA.9-1.tex       asterLA01-20c.tex
asterLA.9-1c.tex      asterLA02-22.tex
asterLA.9-11.tex
```

A real example. The versions of Shaw and Geyer (2010, *Evolution*, doi:10.1111/j.1558-5646.2010.01010.x).

A Plethora of Version Control Systems

Ripped off from Wikipedia (List of version-control software)

Local Only	Client-Server	Distributed
SCCS (1972)	CVS (1986)	BitKeeper (1998)
RCS (1982)	ClearCase (1992)	GNU arch (2001)
	Perforce (1995)	Darcs (2002)
	Subversion (2000)	Monotone (2003)
		Bazaar (2005)
		Git (2005)
		Mercurial (2005)

There were more, but I've only kept the ones I'd heard of.

Don't worry. We're only going to talk about one (git).

Mindshare

Starting from nowhere in 2005, git has gotten dominant mindshare.

In Google Trends, the only searches that are trending up are for git and github.

Searches for competing version control systems are trending down.

Many well known open-source projects are on git. The Linux kernel (of course since Linus Torvalds wrote git to be the VCS for the kernel), android (as in phones), Ruby on Rails, Gnome, Qt, KDE, X, Perl, Go, Python, Vim, and GNU Emacs.

Some aren't. R is still on Subversion. Firefox is still on Mercurial.

Why?

Section 28.1.1.1 of the GNU Emacs manual.

> Version control systems provide you with three important capabilities:
>
> - Reversibility: the ability to back up to a previous state if you discover that some modification you did was a mistake or a bad idea.
> - Concurrency: the ability to have many people modifying the same collection of files knowing that conflicting modifications can be detected and resolved.
> - History: the ability to attach historical data to your data, such as explanatory comments about the intention behind each change to it. Even for a programmer working solo, change histories are an important aid to memory; for a multi-person project, they are a vitally important form of communication among developers.

Getting Git

Type "git" into Google and follow the first link it gives you

`http://git-scm.com/`

Under "downloads" it tells you how to get git for Windows and for Mac OS X.

If you have Linux, it just comes with (use the installer for your distribution).

E. g., on Ubuntu

```
sudo apt-get update
sudo apt-get install git
```

What?

(again from Wikipedia under Git)

> Torvalds sarcastically quipped about the name git (which means "unpleasant person" in British English slang): "I'm an egotistical bastard, and I name all my projects after myself. First 'Linux', now 'git'." The man page describes Git as "the stupid content tracker". The read-me file of the source code elaborates further:
>
> > "git" can mean anything, depending on your mood.

- random three-letter combination that is pronounceable, and not actually used by any common UNIX command. The fact that it is a mispronunciation of "get" may or may not be relevant.
- stupid. contemptible and despicable. simple. Take your pick from the dictionary of slang.
- "global information tracker": you're in a good mood, and it actually works for you. Angels sing, and a light suddenly fills the room.
- "goddamn idiotic truckload of sh*t": when it breaks

Before Anything Else

Tell `git` who you are.

```
git config --global user.name "Charles J. Geyer"
git config --global user.email charlie@stat.umn.edu
```

Of course, replace my name and e-mail address with yours.

Cloning an Existing Project

```
git clone git://github.com/cjgeyer/foo.git
```

Or you can copy an actual repository if it is on the same computer

```
git clone /home/geyer/GitRepos/Git
```

Or you can do the same from another computer

```
git clone geyer@ssh.stat.umn.edu:/home/geyer/GitRepos/Git
```

via `ssh` (requires password or passphrase). The syntax is the same as for remote files when using `scp`.

Starting a New Project

How To

```
mkdir foo
cd foo
git init
```

As yet there are no files in the project, because you haven't put any there. But

```
ls -A
```

shows a directory named `.git` where `git` will store all version control information. (The `-A` flag to the `ls` command is needed to show a file or directory whose name begins with a dot).

A Hint (Git is Woke)

Recent versions of `git` emit the following when doing `git init`

```
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
```

```
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
```

If you do not set the global configuration option you can also do

```
git init -b main
```

(or whatever you like instead of "main") to chose the name of the only branch this repository has at the present time (just after initialization).

Warning about Starting a New Project

If you are making an R package, put the package directory **in** your git repo. Do not make it your git repo. I. e., if the package is `foo`, then the directory `foo` which contains the package (has files `DESCRIPTION`, `NAMESPACE`, etc. and directories `R`, `data`, `man`, etc.) should not be the top level directory of the repo (the one the directory `.git` is in).

This is just my opinion, but following my advice gives you a place to put things related to the package that you do not want to distribute with the package.

Commits

The Index (also called the Staging Area)

Git remembers what you **commit**. Nothing else.

Git offers very precise control of what a commit remembers. It remembers exactly the files you tell it to. Nothing else.

What it remembers in a commit is what you have put in the **index**, also called the **staging area**.

History

Git always remembers every commit and the order in which commits were done.

If you clone a git repository, you get the entire history, all the versions of files remembered in all commits, and the working tree will contain the most recent version of each file. But it does not contain versions never committed.

Git add

The command `git add` adds files to the index (staging area).

Shell file globbing is useful in conjunction with this.

```
git add [A-Z]*
git add R/*.R
git add src/*.{c,h,f,cc}
git add man/*.Rd
git add data/*.{R,tab,txt,csv,rda}
git add tests/*.{R,Rout.save}
git add vignettes/*.Rnw
```

adds most of what you might want to track in an R package.

More on Commits

The command `git commit` does commits.

```
git commit -m "first commit"
```

does a commit. The argument of the `-m` flag is the commit message. If the `-m` flag is omitted, then git drops you into your default editor to compose the commit message.

Commit Message Guidelines

By convention a git commit message consists of

- a short (50 characters or less) line that describes the commit
- followed by a blank line
- followed by a longer explanation that can run on to multiple lines and should be wrapped at 72 characters or so for convenience of reading in standard terminal windows.

As to how to write a good commit message see Section 5.2 of *Pro Git*.

Commit All

```
git commit -a
```

or

```
git commit --all
```

adds all files that are being tracked to the index (staging area) and then does the commit. This is useful when you have made changes to files that were already being tracked (were added in previous commits).

Git Status

The command `git status` (from the man page)

> Displays paths that have differences between the index file and the current HEAD commit, paths that have differences between the working tree and the index file, and paths in the working tree that are not tracked by git (and are not ignored by gitignore(5)). The first are what you would commit by running git commit; the second and third are what you could commit by running git add before running git commit.

Gitignore

The file `.gitignore` in the top-level directory of a repository tells git what to ignore. This file for R package `foo` is

```
*.Rcheck
*.tar.gz
*.bak
```

Note that patterns recognized by the UNIX shell can be used.

You want git to track this file so it is part of every repository.

Git Diff

The command

`git diff`

shows the differences between the working tree and the index (staging area).

The command

`git diff --cached`

between the index (staging area) and the previous commit.

And there are lots more possibilities. The command

`git diff `git log --author=Fred --max-count=1 --format=%H``

shows the diff between the current working tree and the last commit that you were the author (if your name is "Fred").

Help

```
git status --help
man git-status
```

do the same thing (show the man page for `git status`) but

```
git --help
man git
```

are different (the first is terser).

Git Log

The command `git log` describes all commits.

Halftime Summary

```
git config
git clone
git init
git add
git commit
git status
git diff
git log
```

Are all you need to know to work on some projects.

But there is lots more (see `git --help`).

Using GitHub for Backup

If you have a repo on your computer. Make an empty repo of the same name on GitHub clicking on

- Repositories
- New (a button)

and then fill in the form following the instructions to "Skip this step if you're importing an existing repository". Then click "Create Repository" (a button).

Then a new page will be shown on which you want to follow the instructions under "...or push an existing repository from the command line". Follow those instructions and you now have identical repositories on your computer and on github.

The the command

```
git push origin main
```

(or whatever branch you want to push, more on branches in the next section) will move any changes commited on your computer to GitHub. And the command

```
git pull origin main
```

will pull any changes that GitHub has that are not on your computer (this might happen if you work on more than one computer).

Branching and Merging

```
git branch bozo
git checkout bozo
```

switches to a new branch (named "bozo"). Make changes and commits on this branch.

Warning: Never do `git checkout` with uncommitted changes in the working directory. Always do `git commit` before `git checkout` unless `git status` says "nothing added to commit."

```
git checkout main
```

switches to the branch named "main". Make changes and commits on this branch.

The two branches proceed independently from the point of the branch until

```
git checkout main
git merge bozo
```

merges them.

The merge will complete (making a "merge commit") or it may be unable to resolve conflicts (where overlapping changes have been made in the two branches). Then it stops, and you must resolve the conflicts manually, that is, edit files removing the conflict markers and leaving them in the form you want to commit. Then

```
git commit -a
```

does the merge commit (you do not need this step if there were no conflicts).

From `get merge --help`

> Warning: Running git merge with uncommitted changes is discouraged: while possible, it leaves you in a state that is hard to back out of in the case of a conflict.

Never merge unless both branches have all changes committed (`git status` says "nothing added to commit").

Working with Others

Option: All Collaborators Use the Same GitHub Repository

Suppose you

- have a GitHub account,
- have uploaded a public key following the instructions on the course home page or the instructions at GitHub, and

- I have set the `foo` repository at GitHub to allow you write permission.

Then

```
git clone git@github.com:cjgeyer/foo.git
```

uses a different URL from which to clone the repository. If you had already cloned from the read-only URL, then

```
git remote origin git@github.com:cjgeyer/foo.git
```

would change the "origin" to the read-write URL.

Suppose you have made some changes and committed them to your clone.

First get up-to-date with GitHub.

```
git pull origin master
```

This will do a merge and may require resolving merge conflicts (and doing another commit when they are).

Then upload your commits to GitHub

```
git push origin main
```

Warning

Never push changes to a project when you are not up-to-date. You can do this using options to `git push`, but don't. Your collaborators will hate you if you do.

Option: All Collaborators Use Different GitHub Repositories

Alternatively, suppose I don't want you writing to my GitHub repository without my knowledge and hence don't give you write permission.

You can clone my repository on GitHub (making your own GitHub repository) upload your changes there, and either

- you have "forked" my project (o. k., if it has a free software license),
- you can ask me to pull from you and merge the changes, in which case the project is unified again.

In this case aliases for remotes are useful. For example if you are working in repository `foo` then

```
git remote add geyer git@github.com:cjgeyer/foo.git
```

allows you to say

```
git pull geyer master
```

instead of using the full URL.

Git Stash

The command `git stash` is useful for avoiding merge commits.

Suppose you are ready to do a commit and then a push to GitHub.

- You know you have to do a pull before a push.
- You know you cannot do a pull unless all changes have been committed.
- Thus it seems you have to
  - commit
  - pull

- fix merge conflicts
- commit again (merge commit)
- push

in that order. Of course, if there happen to be no merge conflicts to fix, then you do not need the second commit either.

But `git stash` enables another work flow. It hides away any changes you have made to the working directory since the last commit (assuming this was pulled from GitHub). So this work flow is

- `git stash`
- `git pull origin main` [no merge conflicts are possible]
- `git stash pop`
- `git diff` [to see what is different between your work and what was just pulled]
- maybe some editing to get files the way you want them
- `git commit --all`
- `git push origin main`

Second Half Summary

Since the Halftime Summary we have learned about

```
git push
git pull
git branch
git checkout
git merge
git remote
git stash
```

GitHub Not Necessary

If everyone working on a project is in the same organization, then any work flow described can be used without GitHub or any other similar web service.

Many of my git repos are backed up by bare repos on `lts.cla.umn.edu`.

Let us see how to do this. On `lts.cla.umn.edu` we do

```
cd
mkdir -p GitRepos/junk
cd GitRepos/junk
git init --bare
```

Now on my local computer I can do in repo junk

```
git remote add origin ssh://geyer@ssh.stat.umn.edu/home/geyer/GitRepos/junk
```

Now

```
git push -u origin main
```

makes the repo on `lts.cla.umn.edu` have the same contents as the repository on my local computer.

Thus my repo on `lts.cla.umn.edu` serves just as well as GitHub for backup.

Furthermore, if I make this repo on `lts.cla.umn.edu` publicly readable, then anyone who has an account on this computer can pull from it. So the pattern where every collaborator has a different repo can work without GitHub.

Git and RStudio

A quite different set of notes covers using git from within Rstudio.

Zenodo

If you use `git` for reproducible research, then eventually you need to put your get repo in a permanant public repository.

GitHub does not qualify as such. You can completely revise or remove any of your GitHub repos at any time. So it is not permanant.

Zenodo (https://zenodo.org/) is a permanant public repository run by CERN (https://home.cern/) the organization that owns and operates the large hadron collider where the Higgs boson was discovered.

It is possible to link a GitHub public repository to a Zenodo repository.

Here is an example

- GitHub https://github.com/cjgeyer/Evolution-correction
- Zenodo https://zenodo.org/record/7013786

Setting this stuff up is a bit complicated and I did not find the documentation at either site to be helpful.

I actually followed this YouTube video https://www.youtube.com/watch?v=A9FGAU9S9Ow