Git Primer

Jessie C. Runnoe

Introduction

In software development, version control is a class of tools designed to help a team manage changes to code over time. The development of codes using a common user repository is a feature of modern work in large collaborations – it is partly how large surveys such as the Sloan Digital Sky Survey (SDSS) have been able to be so successful.

Git, Mercurial, and SubVersion (SVN) are a few examples of common version control software. They share the practice of keeping track of how codes change as the development team fixes bugs and implements new features. Beyond that, there are some general differences in workflow (a consistent recipe for using version control software to accomplish tasks). SVN maintains a central repository called the trunk from which users check out a working copy. They can then pull changes from other users to update their local copy and commit any subsequent changes that they make. With SVN, the user does not necessarily have the entire repository stored locally. Mercurial and Git are *distributed* version control systems, meaning that each user actually clones the entire repository locally. All copies of the repository are created equally, so users can push and pull their changes among the distributed system as necessary. Another way that Mercurial and Git are different from SVN is that they actually track the *changes*, rather than the files themselves, which saves space. Mercurial and Git have many similarities, but we will be using Git in this course because it is by far the most commonly used version control system in the world.

In this class, all work will be submitted[1] through Git. For instance, homeworks will be issued before the final lecture of a given week and will be due by uploading code and/or documents to our Git repository two weeks later at the start of class (i.e. by Wednesday at 09:30AM).

Every document or piece of code in the class Git repository will be available for everyone in the class to use and edit[2]. If you are worried about other students copying your homework (and other) submissions, note that a simple "diff" in UNIX, and/or logging via "git diff", will make it obvious to me if another student has directly copied your submission. In fact, frequently uploading your code to the Git repository as you write and develop it will make it far harder for your work to be copied than uploading it in a single submission. It is always possible to return a Git repository to an earlier state, so any unwanted changes can be easily redacted.

Version Control with Git

Think of a Git repository as a collaborative directory to which multiple people have access. The directory is special in that it tracks how it changes with time, and logs information on who has made the changes. That way, it's possible for any user who has a clone of the repository to track changes and use anything in the repository. Git is not intended for tracking large data files, so think code and documents.

[1] However, work won't be *graded* through Git, as FERPA frowns on students seeing each other's grades.

[2] Including this one, if you'd like to edit it.

Git has multiple common workflows, we will use a centralized one that is appropriate for small teams. The recipe to track a change in Git with this workflow is to select files that you want to track to a *staging area*. Then, you write a message summarizing the work you have done and commit the changes. Each repository has branches; the main/default one is called the master. Although you won't need to for this workflow, Git users can branch off from the master to organize work efforts (e.g., to implement a feature), and track changes locally. The part of a branch that is selected is called the head and is by default the tip of the master branch, although this can change in more advanced workflows.

Let's make a dummy repository to learn the ins and outs of Git. To get started, you'll need an open terminal window and your favorite text editor. In the terminal, create a directory and change into it. Then issue these commands to initialize your own test repository:

```
git init
```

This command initializes the directory you are in as a Git repository. At some point, if you have never used Git before, it may ask you to define your name and email. You can do that with `git config -global -edit`. Now let's make and commit your first file. I'll pretend I'm making a list of bike trails:

```
echo ‘‘Happy Jack’’ » trails.txt
git add trails.txt
git commit -m ‘‘Added list of trails.’’
```

Now you can open trails.txt in your text editor, update the text a few times, and make several commits. Remember to use `git add` each time to track changes to the staging area and use an informative message with each commit. However, be wary of using `git add *` because it may pick up temporary files.

After I have updated my test file many times, my log of commits looks like this (the top is the most recent):

```
git log -oneline
8fd949c Updated groceries.
38d46d7 Added groceries.
8ab1d45 Updated CO trails.
41bda62 Added CO trails.
efdc3d5 Added WY trails.
2c689a3 Added list of trails.
```

At the end, I started adding groceries to my list of trails. That's a little crazy, so I want to revert back to an earlier version of the trails file. There are several ways to do this in Git, but we will use `git revert` because it maintains a logical file history and is therefore best for collaborative projects. If I use `git revert head`, Git will make a new commit that is the inverse of the last one. This only lets you go one commit back, because executing it a second time reverts the revert. Since I have two commits worth of groceries that I want to revert, I can specify the hash (or ID) of the commit I want to go back to:

```
git revert --no-commit 8ab1d45..head
git commit -m "Reverted to commit before groceries were added to trails list."
```

The `-no-commit` option just lets you use your own more informative commit message instead of the default one generated by `git revert`. Now the crazy grocery changes are still stored in my history (and I could go back and view or adopt them if I want to), but my current working file does not include those changes and the revert is preserved in the history:

```
git log -oneline
461dff4 Reverted to commit before groceries were added to trails list.
8fd949c Updated groceries.
38d46d7 Added groceries.
8ab1d45 Updated CO trails.
41bda62 Added CO trails.
efdc3d5 Added WY trails.
2c689a3 Added list of trails.
```

If you just want to go back and look at the state of your directory several commits ago, you can use `git checkout`. Please be careful to use this only to explore your directory and look at code in the previous state. Commits and branches in this case can result in a detached head, which will complicate a shared history among your classmates.

What about moving, renaming, and deleting files when they're version controlled with Git? Deleting a document is simple, use `git rm` to delete and stage it, and then make a commit noting that you removed that file (alternatively, you can use the UNIX `rm` command and then either `git rm` or `git add` to stage it before the commit). If you want to rename a file, use `git mv` rather than the UNIX command so that Git knows the file has been renamed. If you accidentally use the UNIX `mv` command, you can rename the file back and Git will never know. Play around with some of the tutorials in the links section until you feel comfortable with basic operations in Git.

The Class Repository

Our directory – our Git repository – sits on my home directory on `tomservo` and is called "repos/F20/ASTR8020". The difference between this repository and the one you just practiced on is that this one will be cloned onto all of your classmates computers in a way that can interact with your own. Furthermore, this remote directory is "bare", meaning there is no associated filesystem attached to it. That means that you will clone the class repository from `tomservo`, but you should do this on your own laptop and not while you are logged into `tomservo`.

Our class repository is structured as follows. Beneath the parent directory, which is called `/home/runnojc1/repos/F20/ASTR8020`, I have put a directory:

`ASTR8020/runnoe`.

This directory is divided into weeks of the course (week1, week2, etc.) I will put any useful PDF notes (such as this one) into these directories for the relevant week. The same documents will be linked from the course website. You should similarly create a directory that is your name and beneath it you will create directories for each week of the course (week1, week2, etc.) in which you will work and post your homework submissions:

```
ASTR8020/yourusername
ASTR8020/yourusername/week1
```

To clone our entire remote ASTR8020/ repository to your computer, first create a directory and change into it, then issue this command:

```
git clone runnojc1@vpac01.phy.vanderbilt.edu:/home/runnojc1/repos/F20/ASTR8020
```

This command clones everything in the remote ASTR8020/ repository and stores it on your local machine under the name ASTR8020/. If you look in the directory `ASTR8020/runnoe/week1` you will even see the original of this document (gitprimer.pdf)! Now create a working directory with your name, plus a directory for your week1 tasks:

```
cd ASTR8020/
mkdir runnoe
cd runnoe/
mkdir week1
cd week1/
touch .gitkeep
git add .gitkeep
git commit -m "Added empty week1 working dir with dummy keep file."
```

Note that Git won't let you add an empty directory to your repository, so adding an empty `.gitkeep` file is the industry standard workaround. This commit has only been made to your local version of the repository but has not yet been communicated to your classmates via our remote repository on `tomservo`. In our workflow, it is considered good practice to make sure your repository is completely up to date relative to the remote repo before you push any of your own changes to it. So first, evaluate whether you are out of date:

```
git fetch
git status
```

If your local repository is out of date due to changes made by your classmates, `git status` will tell you. The next step is to pull the latest state of the remote repository to your local clone and merge any changes:

```
git pull
```

The workflow that I will describe for the class will make merging changes very simple since you will never be editing the same files as your classmates. Thus, merging should largely be handled by Git and will be pretty straightforward by following the prompts. Now you are ready to push your changes to the bare repository:

```
git push origin master
```

This will send your updated week1 directory to the bare repository where your classmates will be able to pull the changes. Experiment with this workflow until you are comfortable using it and understand the outputs from `git log` and `git status`. Ask if you have questions.

Common Commands

`git status`
This command prints the status of your repository to the screen. It will tell you about files that are added, tracked, missing, or renamed as well as the state of your local repository clone relative to the remote.

`git fetch`
This command queries the remote class repository to see whether there are any changes since your last pull. Issue it *before pushing any changes to the remote repository.*

`git pull`
This command downloads and merges changes to your local directory to mirror the current copy of the remote repository. Issue this command in combination with `git fetch` *often.*

`git push`
This command sends your commit history and changes to the remote repository. Always issue `git fetch` and `git pull` to merge any changes *before* pushing your own updates to the remote.

`git add anewfile`
This adds a file to the staging area to be included in the next commit. Issue it every time you want to commit a file. Use `git status` to find modified files that are not staged for a commit.

`git commit -m ‘‘this is what I did’’`
This command commits a file you are working on to the local repository. It will be committed to the directory in the repository that mirrors the directory that you are in locally. The `-m` switch commits a comment that will be logged by Git. *Always include a comment.*

`git log`
This command lists all of the changes to the repository. Use `git log -oneline` for a compact, easy-to-read version. To see recent changes, pipe it to `more` at the command line (e.g., `git log | more`). To see changes somebody specific made, `grep` that person's username at the command line (e.g., `git log | grep runnoe -A 3`).

`git ls-files`
This command lists what is in the repository. This is a useful command as you may expect to find a file in the Git repository when, in fact, you forgot to add or commit that file. This command, then, will allow you to see what is actually in the repository as compared to what is in your local directory.

`git diff revA..revB`
This command shows the differences in the repository between commits with hashes A and B. This allows you to track changes that people have made. One use, e.g., would be to see who has added what to a LaTeX document since you went to bed last night.

If you find yourself using many other different commands, feel free to add them to this

document.

Good Practice with Git

There are various examples of good Git etiquette for our workflow to help people share documents and code:

1. Always `git fetch` and then `git pull` before committing anything new to the directory. This way, if somebody else commits something new just before you, then you won't lose track of which version of the directory you're working with.

2. Git is for storing code and documents, *not large data files*. Large data files will make the repository slow to operate. If you have a large data file, keep it local to your machine and share it with people in other ways.

3. Git requires good coding practices. Place comments within the body of your code carefully using your initials. So, my code will have many comment lines of the form, e.g., `# JCR this line of code does this`. When writing documents collaboratively, make similar comments if you make major changes to the document.

4. When you commit a new document using the `git commit` command, *always* provide a comment as in `git commit -m "this is what I did"`.

Class Rules for Git

Do not abuse the collaborative power of Git to plagiarize other student's work. You will learn nothing by copying other people in full. Here are some specific rules designed to allow us to share a collaborative workflow in this course:

1. Do not edit code written by another member of the class without their permission. The reason for this is twofold: a) it maintains separate spaces for each student to submit their independent work and b) it facilitates a workflow that makes merging changes to the repository straightforward. *Editing other people's code that is placed in any directory that contains their name (i.e.* `ASTR8020/theirname/weekx/somecode`*) will be considered grounds for failing the course.*

2. It is permissible to read and to make a copy of any member of the class's code *after* it has been graded as a homework submission...so, in week 2, it will be permissible to raid people's `week1` directory. But, use other people's code by making a copy of it in your personal directory or linking to it in full. *Do not edit it in their directory. Editing other people's code that is placed in any directory that contains their name (i.e.* `theirname/weekd/somecode`*) will be considered grounds for failing the course.* If you have questions about whether it is okay to look at your classmate's code in specific situations, just ask!

3. Provide your own homework solutions. Do not copy each other's work. It is *very* easy for me to check in Git whether your homework submission greatly resembles another

student's submission. *Plagiarizing each other's homework submissions will be considered grounds for failing the course.* Feel free to discuss homework problems and issues with each other but *write your own submissions sitting by yourself.*