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 Tuesday 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

[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.

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.

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.

The Class Repository

Our directory – our Git repository – sits on my home directory on `tomservo` and is called "repos/bare/S25/ASTR8080". This repository 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`. If you want access to the class repository on `tomservo`, then you will need to clone it there as well.

Our class repository is structured as follows. Beneath the parent directory, which is called `ASTR8080/`, I have put a directory `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:

```
ASTR8080/yourusername
ASTR8080/yourusername/week1
```

To clone our entire remote repository to your computer, first create an ASTR8080/ directory and change into it, then issue this command:

```
git clone your_username@vpac01.phy.vanderbilt.edu:/home/runnojc1/repos/bare/S25/ASTR8080
```

Note that you should insert your own username instead of "your_username", but the second instance of my username in the path should remain. This command clones everything in the remote ASTR8080/ repository and stores it on your local machine in the directory that you are in. If you look in the directory `ASTR8080/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 ASTR8080/
```

```
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.

For Windows Users

Windows has a couple of default behaviors that are different than unix-based operating systems that matter for `git`. The first of these is that it uses a different marker for the endings of lines. If you do not manually configure this on your Windows machine, it will cause `git` to always think that every line of a file has changed because the invisible line ending is different. To avoid this in Windows, make sure you are using a `git` install in your Windows Subsystem for Linux (WSL) and Visual Studio Code (VSCode) installation. This will behave like other unix operating systems and avoid the issue.

The second issue to be aware of is that Windows will distinguish between lowercase and capitalized letters in filenames (e.g., files named readme and Readme are different). This is not the case in unix-based operating systems. Thus, you should not use capitalized letters to create distinct filenames because it will confuse the class repository.

Ignoring Files

You can use a `.gitignore` file to tell git extensions, directories, or filenames that you want it to habitually ignore. Git will ignore them in the sense that (a) the `git status` command will not print them as files you may want to stage and (b) if you try to `git add` one, you will see a warning. It is good practice to use a `.gitignore` file, but it will be especially useful in this class because Mac and Python both automatically create files that are problematic to track. I have added a `.gitignore` to our repository, find and inspect it. Also create your own in your working directory, adding any temporary files that your text editor of choice uses.

Be warned that though some common text editors have Git functionality embedded in them in a way that seems very handy, they also sometimes include behaviors you don't expect. This can include staging files that should not go into the repository. Strategic use of your `.gitignore` file can mitigate this issue.

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. It does not download new changes. Issue this in conjunction with `git status` and `git pull` *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 everything in the staging area 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.
`git checkout revA`
You can find revA, the SHA-1 hash for a revision of interest, with a `git log` command. It will be the very long string of characters. This command will then take your repository to a "detached head" state, where you can look around and inspect the state of the repository at that revision. It is not a good idea to edit anything in this situation, but it is useful for looking around (I use it to grade your work at the time it was submitted). You can return to the working repository with `git checkout master`.

`git checkout filename`
If `git status` reveals that a file has been modified, this command will return it to the state of the previous commit. This is useful if you modify a file (e.g., a Jupyter notebook just by opening it) and want to return it to its original state.

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.* `ASTR8080/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 and always reference where it came from. *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.*