Version Control
Statistics 650/750
Week 1 Thursday

Alex Reinhart and Christopher Genovese

30 Aug 2018

If you haven't already, do it **now**:

1. Register at `https://github.com`
2. Visit `https://classroom.github.com/a/MMfm3qKz`
3. Download and install Git: `https://git-scm.com/`
4. Accept the GitHub organization invitation you received (perhaps in the last hour!)

Interlude: Setup Activity

We will use these ideas and commands to set you up for the rest of the semester.

Windows Pre-Setup

First, **Windows users** running Git Bash need to run a script to set up their environment.

To get the url, you can go to `https://github.com/36-750` Navigate to documents > ClassFiles > week1 > setup-profile.py And then hit the "Raw" button. Grab that URL and insert here.

```
cd ~
curl https://raw.githubusercontent.com/36-750/documents/master/ClassFiles/week1/setup-profile
```

Then **exit and restart** Git Bash.

Go Home

Move to your home directory. (How?)

```
cd
```

Create a Class Directory and Move There

Create directories `s750` and `bin` and switch to `s750`.

k #+begin$_{\text{src}}$ sh mkdir s750 bin cd s750 #+end$_{\text{src}}$

The `bin` directory is where you will keep any utility scripts/programs that you want to use regularly.

The `s750` directory is where you will keep all your work for this course throughout the semester.

Setting Up Your Repositories

When you have Git installed and have a GitHub account, do the following. Otherwise, you should do both steps before proceeding, as described in an email from Alex.

If you try to do this later, remember to do it only **after** doing `cd ~/s750` first.

1. Clone the course repositories

   ```
   git clone https://github.com/36-750/documents.git
   git clone https://github.com/36-750/problem-bank.git
   ```

   You will probably need to use your GitHub username and password.

   If you get an error when you run the second command, you need to check your email for the GitHub invitation to join the 36-750 organization, and accept the invitation.

2. Clone your assignment repository. If your github account name is `NNNN`, do

   ```
   git clone https://github.com/36-750/assignments-NNNN.git
   ```

   replacing the NNNN with your account name in the command.

Next Time

When you want to work on your materials, start the shell, and return to your `s750` directory by typing

```
cd ~/s750
```

Version Control

In even a moderate-sized software project, there are many associated files, and they change quickly. Version Control is a way to keep track of those changes so that the history of development can be recovered.

But it does much more. Version control enables:

- working on code simultaneously with other team members
- creating "branches" of code, to try new or experimental features
- distributing code over the web for varied access
- tagging parts of the history for special use
- examining changes in the history in minute detail
- stashing some changes/ideas for later and recovering them

Today we will talk about Git, a popular and powerful distributed version control system. There are dozens of others, most notably Mercurial and Subversion.

Why version control?

One of the key themes of this course is that *code should be understandable to others.* Next week we'll talk about the basic rules of clear, readable, reusable code, such as formatting and style. But there's another piece to it: a reader must understand *why* your code is the way it is. They must be able to deduce its history.

And even if you're not writing for an audience, there's a saying I heard somewhere: The person who knows the most about your code is you six months ago, and you don't reply to email.

So *even when you're working alone*, version control provides many features:

- Keep a complete record of changes. You can always revert back to a previous version or recall what you changed.

- Store a message with every change, so the rationale for changes is always recorded.
- Mark snapshots of your code: "the version submitted to JASA" or "the version used for my conference presentation".
- Easily distribute your code or back it up with a Git hosting service (like GitHub or Bitbucket).

Git concepts

You have a folder with some code (R files, Python files, whatever) in it, plus maybe data, documentation, an analysis report – just about anything. This folder is your *repository*. You'd like to make *snapshots* of this folder: this in the version where I fixed that nasty bug, this is the version where I added new diagnostic plots, this is the version with the updated dataset.

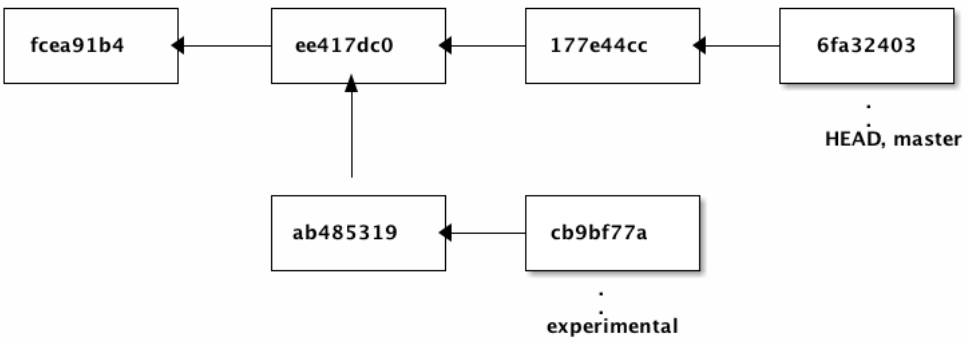
In Git, these snapshots are called *commits*. Each commit is the state of the code at a certain time. Git tracks all the files in your folder to see which have changed and what's eligible to be committed. Every commit has a *parent* – the commit from which it came.

When you're ready to make a snapshot, you tell Git which changes you want to include in the snapshot, and write a *commit message*, describing the snapshot. Git then records it all permanently.

This record of snapshots can be shared with others, who can *clone* your files and snapshots to have their own copy to work with.

Crucially, Git can store parallel histories: if two different people have copies of the same folder and make changes independently, or if one person tries two different ways of changing the same code, snapshots of their work can exist in parallel, and the work can be reconciled ("merged") by Git later.

These parallel lines of development are called *branches*, and can be given names:



Git works by creating a special hidden folder (called `.git/`) inside the folder you're taking snapshots of. All snapshot history, commit information, and other data is stored in this folder. No other websites or services are necessary, though you can *push* your snapshots to a server (we'll return to that later).

Some Git terminology

Git is not particularly friendly to new users. You'll need some terminology:

Repository a directory (local or remote) containing tracked files

Commit a project snapshot

Branch a movable pointer to a commit, usually representing a specific line of development

Tag a metadata object attached to commits

HEAD a pointer to the local branch you are currently on

Tree an object that maps file names to stored data (like file snapshots)

Three important trees:

Working directory the files you see

Index the staging area for accumulating changes

HEAD a pointer to the latest commit

Collaboration in Git

Git is built to allow multiple people to collaborate on code, using this system of snapshots and branches.

Multiple people can have copies of a repository, and can all "push" their separate snapshots to a server, on separate branches of development. These branches can then be *merged* to incorporate the changes made in each.

Sometimes this is easy to do, and Git can do it automatically. Sometimes each branch contains changes to the same parts of files, and Git doesn't know which changes you want to keep; in this case, it asks you to manually resolve the *merge conflict*.

Git-related services

Git repositories can be "pushed" and "pulled" from one computer to another. Various services have sprouted up around this, offering web-based Git hosting services with bug trackers, code review features, and all sorts of goodies. GitHub is the biggest and most popular; Bitbucket is an alternative, and GitLab is an open-source competitor.

We'll use GitHub in this course. You should already have an account set up.

(You can upgrade your GitHub account with a free educational discount at education.github.com, including unlimited private repositories. It's not required for this class, but may be useful to you later.)

Many of these online services provide handy features for collaborating on code. GitHub, for example, has "pull requests": if you've pushed a branch to a repository, you can request that its changes be merged into another branch. GitHub provides a convenient view of all the changes, lets people leave comments, and then has a big button to do the merge. We'll see how to do this shortly.

A simple Git workflow illustrated

Installing Git:

- Mac: Git for Mac (`https://git-scm.com/downloads`) or homebrew -> `brew install git`
- Linux: apt, yum -> `apt install git`, `yum install git`
- Windows: Git for Windows (`https://git-scm.com/downloads`)

1. Set up your configuration. Git records your name and email with each commit:

```
git config --global user.name   "Alex Reinhart"
git config --global user.email  "areinhar@stat.cmu.edu"

git config --list       # check the configs
git config user.name    # ...or just one
```

   Use the email address you used with your GitHub account, so it will recognize you.

   Git also needs to know which editor you like to use. If you installed Visual Studio Code, try

```
git config --global core.editor "code --wait"
```

   For Emacs, use

```
git config --global core.editor emacsclient
```

   If you use some other popular editor, you may need to look this up to find the right command. (Any command-line editor can be used just by putting the command in place of `emacsclient` or `"code --wait"`.) For now, if you use neither, try

```
git config --global core.editor nano
```

2. Clone (download) an existing repository:

```
cd ~/s750
git clone https://github.com/36-750/git-demo.git
```

```
cd git-demo/              # move into the cloned repository
git status                # check the status
ls -a                     # observe the .git hidden directory
```

`git clone` will ask for your GitHub username and password.

3. Open the `git-demo` folder in Explorer or Finder or whatever you use on your computer to find files. Look – it's the same stuff.

4. Make a branch and check it out.

```
git branch your-clever-name-here
git branch
git checkout your-clever-name-here
```

(You can do this in one step with `git checkout -b your-clever-name-here`.)

The branch is split from where you currently are – the commit `git status` shows as most recent.

5. Make some changes to your repository. Add files, edit something, whatever. In Git terminology, you're making changes in the *working directory*.

6. Add the changes to the *index*, so they are staged to be committed.

```
git status

git add file_you_changed.py
```

7. Commit the changes (No, no, I'm sane, I tell you. SANE!)

```
git commit
```

Git will open an editor to let you type a full commit message. Close the file when you're done so Git knows you're done.

If you have a very short commit message, you can do it in one step:

```
git commit -m "Your very short commit message"
```

8. Make more changes and stage them.
9. Look at differences

```
git log                                    # default log
git log --oneline --abbrev-commit  # terser log
git diff 3597a84 e3f8f5d
```

10. Push this branch to the remote repository (on GitHub)

```
git push --set-upstream origin your-clever-name-here
```

    The `--set-upstream` option is only necessary once, to tell Git that the "upstream" for this branch – the remote location for it – is the corresponding branch on GitHub, which will be created automatically.

11. Switch back to the main branch:

```
git checkout master
ls
```

    Now look at the files in your repository. Notice they've all changed back to what they looked like *before* you switched to your branch.

12. Look at the status

```
git status
git log --pretty=oneline --abbrev-commit
```

13. Make a pull request on GitHub

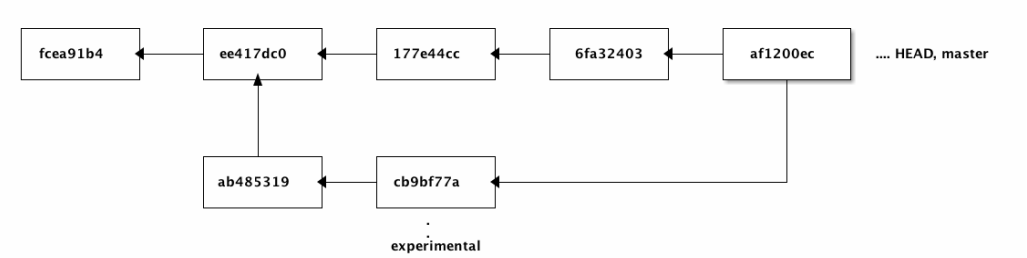
    `https://github.com/36-750/git-demo`

This is similar to the workflow we'll use for submitting homework. We'll provide you a script which automates part of it, but it's good to know basic Git operations so you know what's going on when you switch branches and submit homework.

An activity

Let's go back to that commit graph we were looking at. Suppose we merged the `experimental` branch into `master`:

```
git checkout master
git merge experimental
```



The commit `af1200ec` is a *merge commit*, the snapshot of the code after `experimental` has been merged to `master`. The merge commit has two parents.

Git can show you a graph of commits with the command

```
git log --graph --oneline --color
```

I'd like you to recreate the graph above by making commits and branches. Team up with someone next to you to plan how to do this. Start from the `master` branch:

```
git checkout master
```

To make a graph just starting from master, not including previous commits, try

```
git log --graph --oneline --color e625475.. --
```

which tells Git to graph the commits since `e625475`, the most recent I made on the `master` branch. It should be empty at first, since there is nothing new on `master`.

Good Git habits

- Write good commit messages!

  Commit messages should explain what you changed and *why* you changed it, so you understand your code later, and collaborators understand the purpose of your changes. An example Git log entry:

  ```
  commit c52d398a97ba4a4c933945d3045cd69cbf7f9de0
  Author: Alex Reinhart <areinhar@stat.cmu.edu>
  Date:   Tue May 24 16:31:23 2016 -0400

      Fix error in intensity bounds calculation

      We incorrectly assumed that if the query node entirely preceded the
      data node, the bounds had to be zero. This is not the case: the
      background component is constant in time.

      Instead, have t_bounds return negative bounds when this occurs, which
      cause the foreground component only to be estimated as zero.

      Update kde_test to remove the fudge factor and instead test based on
      the desired eps, not the (max - min) reported by tree_intensity. I
      suspect rounding problems mean the (max - min) is sometimes zero,
      despite the reported value being very slightly different from what
      it should be.
  ```

  Notice:

  - The first line is a short summary (subject line)
  - There are blank lines between paragraphs and after the summary line
  - The message describes the reasoning for the change
  - The message is written in the imperative mood ("Fix error" instead of "Fixed error")

- More advanced branching

  If you want to try a new algorithm, reorganize your code, or make changes separately from someone else working on the same code, make a branch.

  We used a branch above to make a pull request. But what if the `master` branch is edited at the same time we're working on our own separate branch?

  Fundamental branch operations: merging and rebasing.

1. Make another change on `master` and commit it
2. Back to your branch

```
git checkout your-clever-name-here
```

3. Rebase on `master`

```
git rebase master
```

4. Check the log now – we have all of master's commits.
5. Alternately, we can merge instead of rebasing. Merging takes another branch's changes, runs them on the current branch, and saves them as a new commit.

```
git checkout master
git merge your-clever-name-here
```

   In this case, the merge was a "fast-forward": there were no other changes in `master`, so no merge commit was necessary.

- Use git blame to find out why changes were made If you ever wonder why a line of code is the way it is, use git blame.

```
git blame fileHomework.py
```

  Use "git show" to see the responsible commit.

RStudio

For those of you using RStudio, there is a handy graphical interface built in, once you create an RStudio project for your work. We recommend learning the command-line basics first, however.

Resources

Some links to helpful resources:

- TryGit has a simple interactive introduction
- Git: The Simple Guide

- The Software Carpentry Git lesson
- ProGit
- Git Reference Manual
- RStudio Git integration documentation

Git has extensive (but not always helpful) manual pages, e.g. `man git-merge`