# Version Control
# Statistics 650/750
# Week 1 Thursday

Alex Reinhart and Christopher Genovese

31 Aug 2017

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

## Course Procedure

### Resources

- All course materials and course work will be accessed through `GitHub`. Instructions were emailed to you; talk to us if you didn't get them.
- You should have received an email inviting you to the Students group on GitHub. Please accept it. This gives you access to the course materials.

  Key course materials:

  | Repository | Contents |
  |---|---|
  | `https://github.com/36-750/documents` | Announcements, lecture notes, data, and other resourc |
  | `https://github.com/36-750/problem-bank` | Assignment descriptions and associated files and data |
  | `https://github.com/36-750/git-demo` | A supplementary demo for today's class |

- Several useful books will be available for you to borrow anytime. They are listed in the `documents` repository and are available (for the moment) from Professor Genovese. *Do not hesitate to ask for them.*

- The instructors will be available to meet to answer your questions. Scheduling of office hours will be data-driven.
- You will interact with the TAs mostly through discussions on GitHub, but they will also be available to answer questions about their feedback. They may hold fixed hours or be available by appointment depending on need.

## Assignments

Details on assignments are spelled out in the syllabus. **Read the Syllabus!**

The work in this course consists of programming and related exercises, including revision of your own programs. There are no exams.

Assignments are drawn from a large repository of problems that vary in length, complexity, topic, and scope. This repository is available on GitHub via `https://github.com/36-750/problem-bank` and will be expanded as the semester proceeds.

You can choose which assignment to do each week, out of those in the problem bank. (Occasionally we will require a specific assignment or set of assignments.)

### Revisions

You will submit assignments and revisions, receive feedback from the TAs, and ask questions about that feedback using "pull requests" on your personal repository. We will describe the procedure in detail today.

Because revision is an important part of the development process, we allow you to revise (and refine based on feedback) your submitted assignments. Each assignment or challenge project may be revised at most twice.

Note that feedback from the TA is contingent on meeting the basic requirements for submission. So for example, if required tests or scripts are missing or if there are significant deficiencies with coding style and organization, the review will stop at noting that issue and you will have used a submission/revision for little gain.

You can submit an assignment at most twice per week, and revisions to a previous submission count. This gives you 28 opportunities during the semester to submit an assignment. There will also be two scheduled challenges during the semester, each of which will extend over more than a week. When planning your submissions, please remember that you will need to spend time on the challenges.

## Grading

Each exercise in a stand-alone or vignette assignment is graded as either "Mastered" or "Not yet mastered." Challenges are graded as "Sophisticated," "Mastered," or "Not yet mastered." The meaning of these classifications and the criteria required to achieve them is spelled out in the course grading rubric, which is available in the documents repository. Each challenge will have its own specialized rubric.

Key points:

- It's OK to make mistakes.
- Try to find ways to expand your skills and perspectives.
- Ask for help if you need it, from us and your peers.
- Pay attention to the rubric.
- Practice, revise, repeat.
- Challenge yourself.

Grades for the course are determined by the number of "Mastered" and "Sophisticated" marks that you receive, with adjustments for participation and other considerations. See the syllabus for the specifics.

Note you are therefore **not penalized** for 'Not yet mastered" grades, and the purpose of revisions is to improve your programs.

### Request

**Please put `[650]` or `[750]` in your email subject lines!**

## 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 Chris will 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. 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.

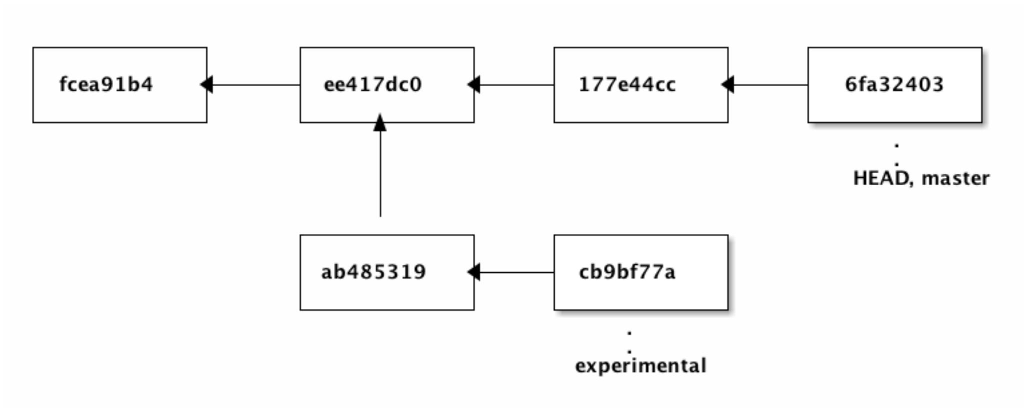
In Git, these snapshots are called *commits.* Each commit is the state of the code at a certain time. Git tracks all the files in your folder to see which have changed and what's eligible to be committed.

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** an object that embodies a project snapshot

**Branch** a movable pointer into a list of commits, usually representing a separate line of development

**Tag** a metadata object attached to commits

**HEAD** a pointer to the local branch you are currently on

**Blob** a piece of stored data, typically a snapshot of a file

**Tree** an object that maps file names to blobs

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 both "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 does 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

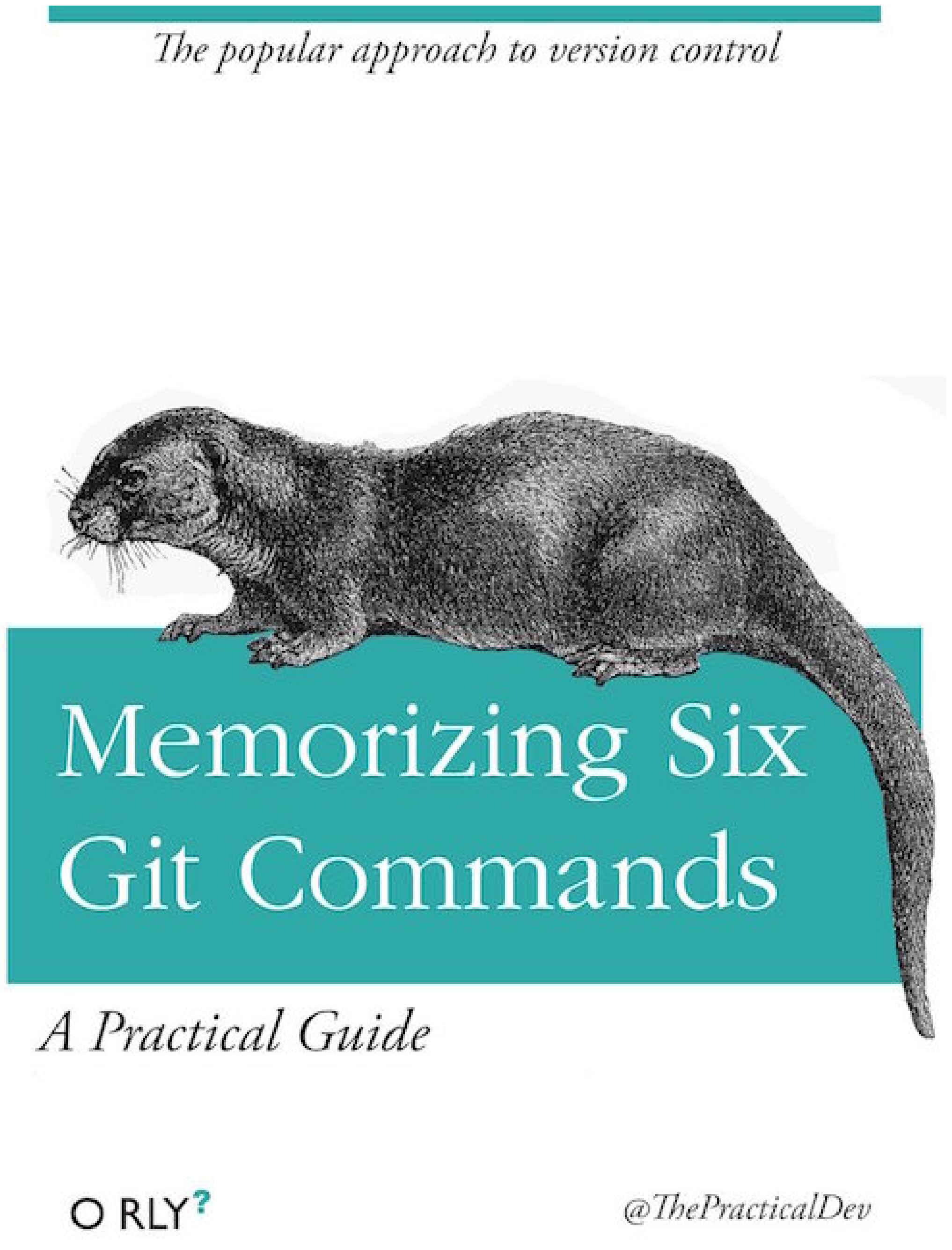
Git repositories can be "pushed" and "pulled" from one computer to another. Various services have sprouted up around this, offering web-based Git hosting services with bug trackers, code review features, and all sorts of goodies. GitHub is the biggest and most popular; Bitbucket is an alternative, and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


The popular approach to version control
Memorizing Six Git Commands
A Practical Guide
O RLY?
@ThePracticalDev

Installing Git:

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

2. Clone (download) an existing repository:

   ```
   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. 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.

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

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

```
git status

git add file_you_changed.py
```

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

```
git commit -m "This is a detailed description of my changes"
```

   If you don't use the `-m` option, Git will open an editor to let you type a full commit message.

7. Make more changes and stage them.

8. Look at differences

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

9. Another commit, and add some tags

```
git tag -a v0.1.0 -m "First working release"
```

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. Back to the main branch

```
git checkout master
```

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`

## 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) man pages, e.g. `man git-merge`

## Plan going forward

### Submit homework assignments as GitHub pull requests

Start your homework assignments on a new branch. Create a folder in your repository for each assignment's code (named meaningfully, like "git-game" or "max-sub-sum").

Once you've finished your assignment and committed your changes, go to GitHub and open a pull request. (Click "Pull requests" on the right and hit the big green "New pull request" button.) Select the branch you'd like to submit and enter a title and description.

Full instructions are given in the `README.org` file in your assignment repository.

We'll leave comments on your code when we grade.

### Homework

This week, you can start work on any exercise in the `git` tag in the problem bank. If you're not familiar with Git, I recommend starting with the Git Game problem.