Version control

with git and GitHub

Karl Broman

Biostatistics & Medical Informatics, UW–Madison

biostat.wisc.edu/~kbroman
github.com/kbroman
@kwbroman
Course web: bit.ly/tools4rr

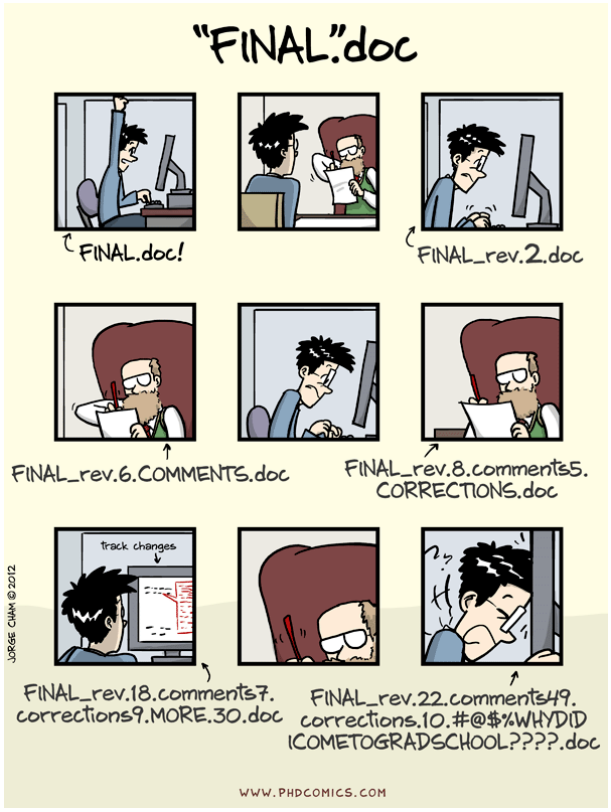
Slides prepared with Sam Younkin

Version control is not strictly necessary for reproducible research, and it's admittedly a lot of work (to learn and to use) in the short term, but the long term benefits are enormous.

The advantages are: you'll save the entire history of changes to a project, you can go back to any point in time (and see what has changed between any two points in time), you don't have to worry about breaking things that work, and you can easily merge changes from multiple people.

I now use version control for basically everything: software, data analysis projects, papers, talks, and web sites.

People are more resistant to version control than to any other tool, because of the short-term effort and the lack of recognition of the long-term benefits.



http://www.phdcomics.com/comics/archive.php?comicid=1531

This is typical. And never use “final” in a file name.

Methods for tracking versions

- Don't keep track
  - good luck!
- Save numbered zip files
  - Unzip versions and `diff`
- Formal version control
  - Easy to study changes back in time
  - Easy to jump back and test

There are three methods for keeping track of changes: don't keep track, periodically zip/tar a directory with a version number, or use formal version control.

Imagine that some aspect of your code has stopped working at some point. You know it was working in the past, but it's not working now. How easy is it to figure out where the problem was introduced?

Why use formal version control?

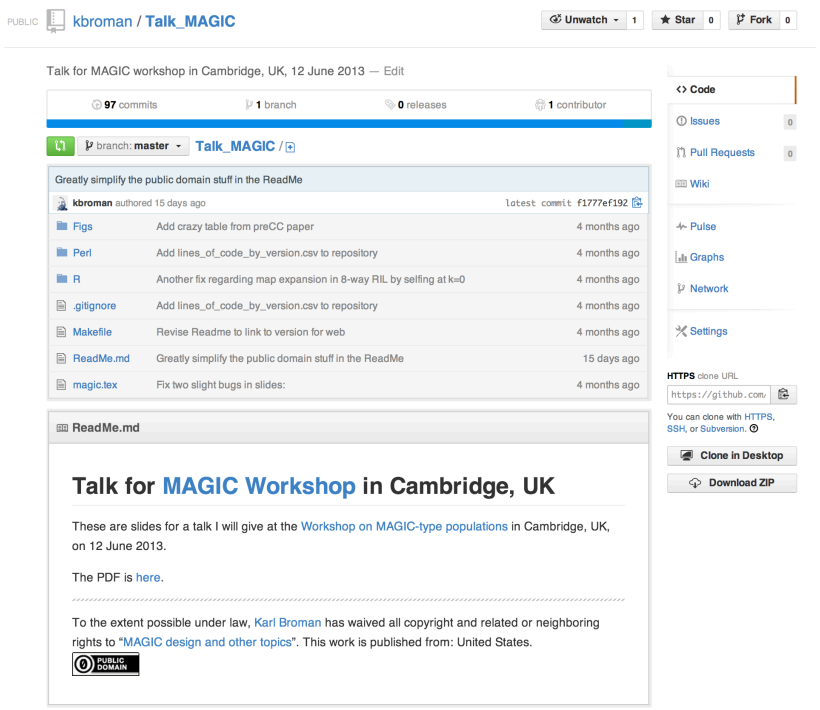
- History of changes
- Able to go back
- No worries about breaking things that work
- Merging changes from multiple people

With formal version control, you'll save the entire history of changes to the project, and you can easily go back to any point in the history of the project, to see how things were behaving at that point.

You'll be able to make modifications (e.g., to try out a new feature) without worrying about breaking things that work.

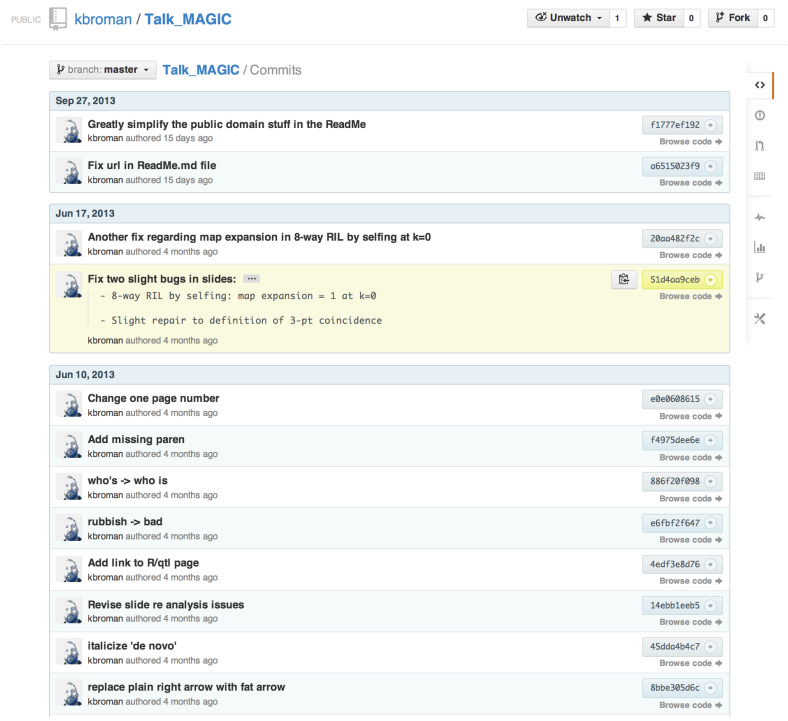
And version control is especially useful for collaboration. If a collaborator has made a bunch of changes, it'll be much easier to see what was changed and to incorporate those changes.

Example repository



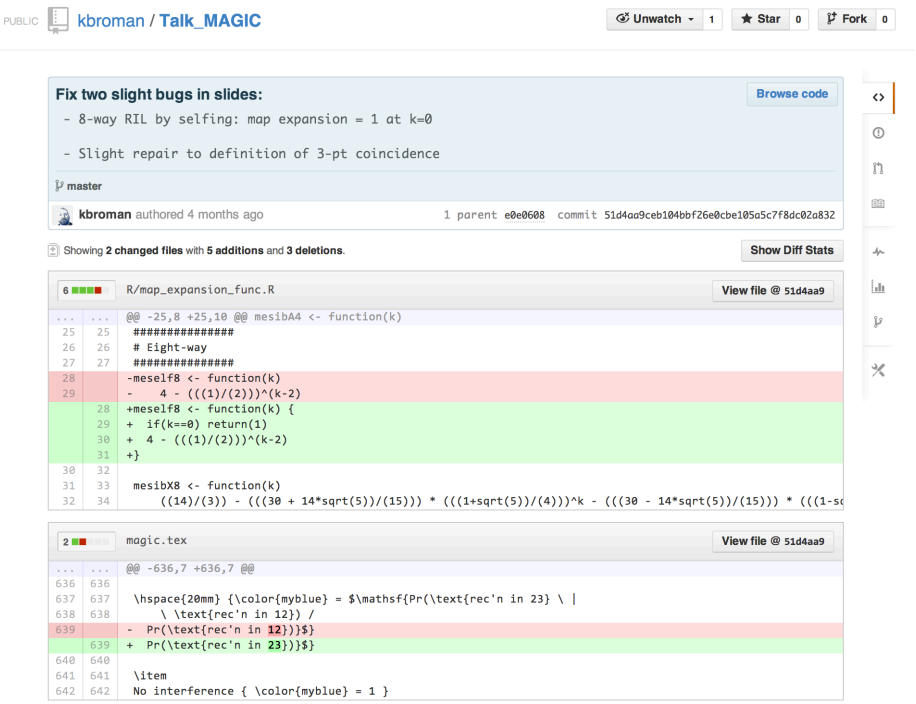
This is a snapshot of a repository on GitHub: a set of files and subdirectories with more files. You can easily explore the contents.

Example history



This is a short view of the history of changes to the repository: a series of "commits."

Example commit



This is an example of one of those commits, highlighting what lines were added and what lines were removed.

What is git?

- Formal version control system
- Developed by Linus Torvalds (developer of Linux)
  - used to manage the source code for Linux
- Tracks any content (but mostly plain text files)
  - source code
  - data analysis projects
  - manuscripts
  - websites
  - presentations

We're going to focus on git, the version control system developed by Linus Torvalds for managing the source code for Linux.

You can track any content, but it's mostly for tracking plain text files, but that can be most anything (source code, data analysis projects, manuscripts, websites, presentations).

Why use git?

- It's fast
- You don't need access to a server
- Amazingly good at merging simultaneous changes
- Everyone's using it

Git is fast, you can use it locally on your own computer, it's amazingly good at merging changes, and there are lots of people using it.

What is GitHub?

- A home for git repositories
- Interface for exploring git repositories
- Real open source
  - immediate, easy access to the code
- Like facebook for programmers
- Free 2-year "micro" account for students
  - github.com/edu
- (Bitbucket.org is an alternative)
  - free private repositories

GitHub is a website that hosts git repositories, with a nice graphical user interface for exploring git repositories.

Source code on GitHub is real open source: anyone can study it and grab it.

GitHub is sort of like Facebook for programmers: you can see what people are up to, and easily collaborate on shared projects.

It's free to have public repositories on GitHub; if you want private repositories, you generally have to pay, but I understand that students can get a two-year account that allows 5 private repositories.

Bitbucket.org is an alternative; it allows unlimited private repositories. I'm cheap, so I use Bitbucket for my private repositories.

Why use GitHub?

- It takes care of the server aspects of git
- Graphical user interface for git
  - Exploring code and its history
  - Tracking issues
- Facilitates:
  - Learning from others
  - Seeing what people are up to
  - Contributing to others' code
- Lowers the barrier to collaboration
  - "There's a typo in your documentation." vs. "Here's a correction for your documentation."

GitHub takes care of the server aspects of git, and you get a great GUI for exploring your repositories.

GitHub is great for browsing others' code, for learning; you don't even have to download it to your computer. And it's really easy to contribute to others' code (e.g., to report typos in their documentation).

Basic use

- Change some files
- See what you've changed

```
git status
git diff
git log
```

- Indicate what changes to save

```
git add
```

- Commit to those changes

```
git commit
```

- Push the changes to GitHub

```
git push
```

- Pull changes from your collaborator

```
git pull
git fetch
git merge
```


These are the basic git commands you'll use day-to-day.

`git status` to see the current state of things, `git diff` to see what's changed, and `git log` to look at the history.

After you've made some changes, you'll use `git add` to indicate which changes you want to commit to, and `git commit` to commit to them (to add them to the repository).

You use `git push` to push changes to GitHub, and `git pull` (or `git fetch` and `git merge`) to pull changes from a collaborator's repository, or if you're synchronizing a repository between two computers.

Initialize repository

- Create (and `cd` to) a working directory
  - For example, `~/Docs/Talks/Graphs`
- Initialize it to be a git repository
  - `git init`
  - Creates subdirectory `~/Docs/Talks/Graphs/.git`

```
$ mkdir ~/Docs/Talks/Graphs
$ cd ~/Docs/Talks/Graphs
$ git init
Initialized empty Git repository in ~/Docs/Talks/Graphs/.git/
```


If you're starting a new, fresh project, you make a directory for it and go into that directory, and then you type `git init`. This creates a `.git` subdirectory.

Produce content

- Create a `README.md` file

```
## Talk on &ldquo;How to display data badly&rdquo;

These are slides for a talk that I give as often as possible,
because it's fun.

This was inspired by Howard Wainer's article, whose title I
stole: H Wainer (1984) How to display data badly.
American Statistician 38:137-147

A recent PDF is
[here](
http://www.biostat.wisc.edu/~kbroman/talks/graphs2013.pdf).
```


Start creating a bit of content, such as a Readme file. You can use Markdown to make it look nicer.

Incorporate into repository

- Stage the changes using `git add`

```
$ git add README.md
```


Use `git add` to tell git that you want to start keeping track of this file. This is called "staging," or you say the file is "staged."

Incorporate into repository

- Now commit using `git commit`

```
$ git commit -m "Initial commit of README.md file"
[master (root-commit) 32c9d01] Initial commit of README.md file
 1 file changed, 14 insertions(+)
 create mode 100644 README.md
```

- The `-m` argument allows one to enter a message
- Without `-m`, `git` will spawn a text editor
- Use a meaningful message
- Message can have multiple lines, but make 1st line an overview

Use `git commit` to add the file to the repository.

Using git on an existing project

- `git init`
- Set up `.gitignore` file
- `git status` (did you miss any?)
- `git add .` (or name files individually)
- `git status` (did you miss any?)
- `git commit`

I recommend using git with all of your current projects. Start with one.

Go into the directory and type `git init`. Then use `git add` repeatedly, to indicate which files you want to add to the repository.

Then use `git commit` to make an initial commit.

Removing/moving files

For files that are being tracked by git:

Use `git rm` instead of just `rm`
Use `git mv` instead of just `mv`

```
$ git rm myfile
$ git mv myfile newname
$ git mv myfile SubDir/
$ git commit
```


For files that are being tracked by git: If you want to change the name of a file, or if you want to move it to a subdirectory, you can't just use `mv`, you need to use `git mv`.

If you want to remove a file from the project, don't use just `rm`, use `git rm`. Note that the file won't be completely removed; it'll still be within the history.

A few points on commits

- Use frequent, small commits
- Don't get out of sync with your collaborators
- Commit the sources, not the derived files
  (R code not images)
- Use a `.gitignore` file to indicate files to be ignored

```
*~
manuscript.pdf
Figs/*.pdf
.RData
.RHistory
*.Rout
*.aux
*.log
*.out
```


I recommend using frequent, small commits. I'll make a batch of changes with a common theme, make sure things are working, then add and commit.

In projects with collabotors, be sure to pull any changes from them before starting to make your own changes, and encourage your collaborators to do the same. If you both make a month's changes in parallel, merging the changes will be harder.

I commit only the source, and not files that are derived from those sources. For a manuscript, though, I might include the pdf at major milestones (at submission, after revision, and upon acceptance), so that I don't have to work as hard to reconstruct them.

Use a `.gitignore` file so that untracked files don't show up with `git status`. You can have a global ignore file, `~/.gitignore`.

But leaving off critical files is a common mistake.

First use of git

```
$ git config --global user.name "Jane Doe"
$ git config --global user.email "janedoe@wisc.edu"

$ git config --global color.ui true

$ git config --global core.editor emacs

$ git config --global core.excludesfile ~/.gitignore_global
```


The very first time you use git, you need to do a bit of configuration.

All of this stuff gets added to a `~/.gitconfig` file

Getting started with GitHub

- Get an account
- Set up ssh keys
  - Look for files `~/.ssh/id_rsa` and `~/.ssh/id_rsa.pub`
  - `ssh-keygen -t rsa -C "janedoe@wisc.edu"`
  - Copy contents of `~/.ssh/id_rsa.pub`
- Add SSH key at GitHub
  - Account settings
  - SSH Keys
  - Add SSH key
  - Paste contents of `~/.ssh/id_rsa.pub`
- Similar thing at BitBucket

To use GitHub, you first need to set up an account. Use the email that you used with git.

To push to your GitHub repositories, you need to you as password so that GitHub knows it's you. I use ssh, so that I don't have to type in my password every time.

The procedure is similar at Bitbucket.

Set up GitHub repository

- Click the "Create a new repo" button
- Give it a name and description
- Click the "Create repository" button
- Back at the command line:

```
git remote add origin git@github.com:username/repo
git push -u origin master
```


To create a GitHub repository, I generally first set things up locally (using `git init` and then a bit of `git add` and `git commit`).

Then go to GitHub and click the “Create a new repo” button. Give it a name and description and click “Create repository.”

The back at the command line, you use `git remote add` to indicate the github address; then `git push` to push everything to GitHub.

Configuration file

Part of a `.git/config` file:

```
[remote "origin"]
url = git@github.com:kbroman/qtl.git
fetch = +refs/heads/*:refs/remotes/origin/*

[branch "master"]
remote = origin
merge = refs/heads/master

[remote "brian"]
url = git://github.com/byandell/qtl.git
fetch = +refs/heads/*:refs/remotes/brian/*
```


The `git remote add` commands adds stuff to the `.git/config` file; if you've made a mistake, you can just edit this file.

Branching and merging

- Use branches to test out new features without breaking the working code.

```
git branch devel
git branch
git checkout devel
```

- When you're happy with the work, merge it back into your master branch.

```
git checkout master
git merge devel
```


Branching is a really important feature of git. Create a branch to test out some new features without breaking your working software.

`git branch` is used to create branches and to see what branches you have.

`git checkout` is used to switch among branches.

`git merge` is used to merge a different branch into your current one.

Issues and pull requests

- Problem with or suggestion for someone's code?
  - Point it out as an Issue
- Even better: Provide a fix
  - Fork
  - Clone
  - Modify
  - Commit
  - Push
  - Submit a Pull Request

One of the best features of GitHub is the ease with which you can suggest changes to others' code, either via an Issue, or best of all via a Pull Request.

Suggest a change to a repo

- Go to the repository:
  `http://github.com/someone/repo`
- Fork the repository
  Click the "Fork" button
- Clone your version of it
  `git clone git@github.com:username/repo`
- Change things locally, `git add`, `git commit`
- Push your changes to *your* GitHub repository
  `git push`
- Go to *your* GitHub repository
- Click "Pull Requests" and "New pull request"

To suggest a change to someone's repository, go to their repository and click the "Fork" button. This makes a copy of the repo in your part of GitHub.

Then go back to the command line and `clone` your version of the repository.

Make changes, test them, `add`, and `commit` them, and `push` them to your GitHub repository.

Then go back to your GitHub repository and click "Pull Requests" and "New pull request."

Pulling a friend's changes

- Add a connection

```
git remote add friend git://github.com/friend/repo
```

- If you trust them, just pull the changes

```
git pull friend master
```

- Alternatively, fetch the changes, test them, and *then* merge them.

```
git fetch friend master
git branch -a
git checkout remotes/friend/master
git checkout -b friend
git checkout master
git merge friend
```

- Push them back to your GitHub repo

```
git push
```


If a friend (or perhaps someone you don't even know) has made suggested changes to your repository by a Pull Request, you'll get an email and it will show up on your GitHub repository.

On the command line, use `git remote add` to make a connection to their repository.

Then use `git pull`, or (better) use `git fetch`, test them out, and then use `git merge`.

Finally, `push` the changes back to your GitHub repository.

Merge conflicts

Sometimes after `git pull friend master`

```
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Automatic merge failed; fix conflicts and then commit the result.
```

Inside the file you'll see:

```
<<<<<<< HEAD
A line in my file.
=======
A line in my friend's file
>>>>>>> 031389f2cd2acde08e32f0beb084b2f7c3257fff
```

Edit, add, commit, push, submit pull request.

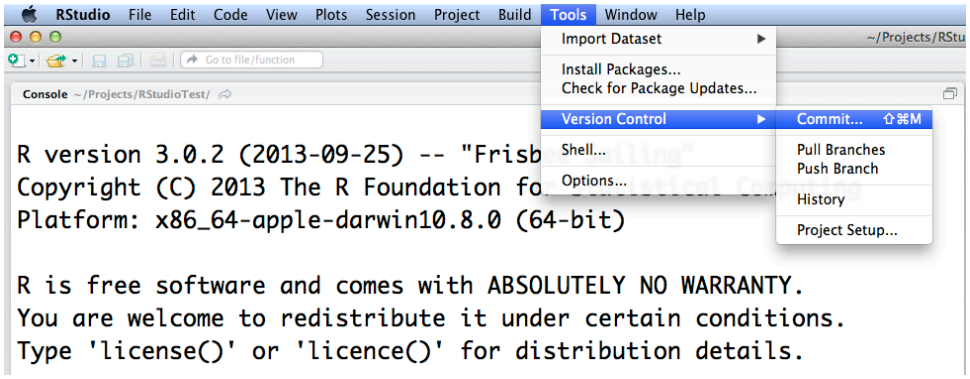
Sometimes there will be conflicts: you and your collaborator will have been making changes to the same portion of a file and you'll have to resolve the differences.

It's perhaps surprising how seldom this happens. git is really good at merging changes.

If there's a merge conflict, there'll be a big warning message on `git pull` or `git merge`, When you open the offending file in an editor, look for lines with `<<<<<<<`, `=======`, and `>>>>>>>`. Pick and choose and make the file just as you want it.

Then, `git add`, `git commit`, and `git push`.

git/GitHub with RStudio



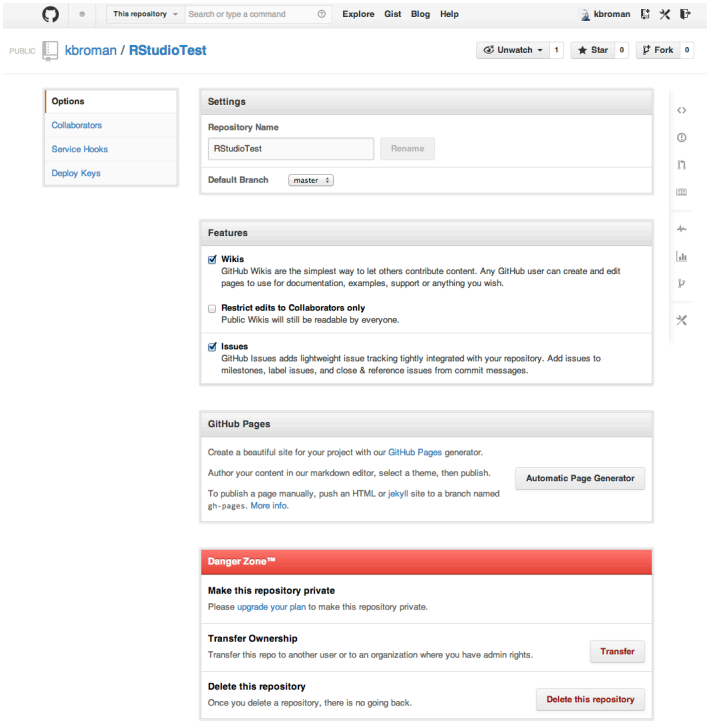
See GitPrimer.pdf or RStudio page

RStudio has great features for using git and GitHub.

I'm not going to spend time talking about this here; google `git site:rstudio.com`.

The key thing is that a Project in RStudio is a directory (with some RStudio configuration file, `blah.Proj`) and will be your git repository.

Delete GitHub repo



To learn git and GitHub, you'll want to create some test repositories and play around with them for a while. You may want to delete them later.

On your computer, if you delete the `.git` subdirectory, it'll no longer be a git repository.

On GitHub, go to the settings for the repository and head down to the Danger Zone.

Git at Statistics, UW-Madison

- Easy to use, free infinite private repositories.
- Not as nice of interface to review code: Rely on GUI or private web page.
- When your ssh account expires, your access to them expires.

If you have an account on the UW-Madison Statistics server, you can use git there in place of GitHub.

The advantage is that you can have as many private repositories as you want.

The disadvantages are that you won't have the GitHub interface and you can only use this as long as you have a Statistics account.

I haven't done this myself; these three slides were kindly provided by Tim Grilley.

Git at Statistics, UW-Madison

Setup (on server):

- Connect to server

  ```
  ssh bigmem01.stat.wisc.edu
  ```

  Consider using kinit + aklog if logging on frequently
- Make Folder

  ```
  cd Repositories
  mkdir NewRepository
  ```
- Initialize Server Repository

  ```
  cd NewRepository
  git init
  ```


To set up a repository you just log in to one of the Statistics computers, create a directory, and use `git init`.

Git at Statistics, UW-Madison

Usage (on client, e.g. laptop):

- Clone/Pull onto other systems

```
git clone ssh:\\bigmem01.stat.wisc.edu\~[user]\Repositories\NewRepository
```

- Make changes, and commit

```
git add -i
git commit -m 'An informative message here.'
```

- Push changes back

```
git push origin
```


This is what you'd do on your local computer (e.g., a Windows laptop).

On a Mac, you'd need to replace the backslashes with forward slashes.

Open source means everyone can see my stupid mistakes.

Version control means everyone can see every stupid mistake I've ever made.

bit.ly/stupidcode

If you store your code on GitHub, everyone can see everything. They can even see everything that ever was.

I think this openness is a Good Thing. You may be shy about your code, but probably no one is looking. And if they are looking, that is actually a Good Thing.