HOW WE GIT

GEOFF RYAN (NYU)

This is how we git.

1. Why we `git`

1.1. **What is `git`?** `git` is Version Control Software, software that tracks, remembers, and restores data over time. Version Control Software (VCS) is incredibly important for any long term project. At the very least, it can act as a backup in the case of accidental data loss (a failed disk or a misplaced `rm -rf *`). Day to day it serves as a way to look back to older versions of your project, which can be very useful for testing and restoring lost functionality. At its best, VCS encourages you to make changes to your project, sometimes sweeping ones, secure in the knowledge you can always restore to a previous version if something breaks. A good VCS platform will also allow you to easily share your project on multiple computers, keeping all versions up to date to the latest changes with a minimal effort on the user's part.

There are several VCS platforms in use currently. `git` is the best one. It is built on knowledge gained from older platforms like SVN, and it has `Github`. A `git` project is called a "repository," or "repo" for short.

1.2. **What is `Github`?** `Github` is a website that hosts remote `git` repos. It is free to use and provides a great deal of functionality for sharing, merging, and forking code bases. It provides all repositories with a webpage, wiki page, and forum for tracking issues and conversations between developers. It is incredibly useful.

1.3. **So?** The combination of `git` and `Github` is incredibly powerful. Multiple users can work on the same repository, keep local copies up to date on multiple machines, and consistently follow up on issues and edits. At the very least, it is a very reliable backup system and way to keep code up to date on several computers.

There is a deeper reason for this too: scientific reproducibility. As scientists, it is our responsibility to ensure all our results are replicable and reproducible. All of our codes are constantly in a state of development, we're modifying the source code all the time to run new simulations or refine old ones. Sometimes new versions of a code won't replicate old results. With `git` and `Github` you always have access to older versions of your code, so it is always possible to reproduce any result you have made. It is possible to do this with some other method (save an old version of your code in another folder, or on a website). It is possible to keep backups of your code (on an external hard drive). It is possible to move your code by `scp` and e-mail. `git` and `Github` allow you to do all of these things with a single workflow, which is really where their power lies. So, we should all use `git` for everything, all the time!

2. Installation and Configuration

`git` is available from `https://git-scm.com/`, and is included in package managers for Linux and OSX. Installation is very standard, and should give you no trouble.

After installation, you want to configure `git` so that it knows who you are. All edits to projects, called "commits," are tagged with your name and email. To make sure the correct ones are used, run:

```
$ git config --global user.name "Johannes Kepler"
$ git config --global user.email johannes.kepler@gmail.com
```

Make an account on `Github` (`github.com`). The email should probably be the same email you configured `git` with. Give your public SSH keys to `Github` by following the instructions here: (`https://help.github.com/articles/generating-ssh-keys/`). Do this on every machine you want to have access to your project, including remote servers. This allows you to push and pull code from these machines back to the `Github` servers.

You are set up and ready to use `git` like a pro! Or at least like a rank amateur like me.

3. Useful Web References

Instead of going through a discussion of each command, we'll link here useful web resources. Much has been written about how to use `git` and `Github`, no need to reinvent the wheel.

3.1. **Hello World.** A very succinct introductory tutorial that gets you doing the fundamental tasks on `Github` in 10 minutes is:
`https://guides.github.com/activities/hello-world/`

3.2. **Pro Git.** This book is a comprehensive guide to `git`. It explains the underlying structure of the version control system utilized by git and how each command is used. It is available digitally for free at:
`https://git-scm.com/book/en/v2`.

3.3. **Cheat Sheets.** Useful quick references. Both of these seem pretty solid, pick one you like!
`http://www.git-tower.com/blog/git-cheat-sheet/`
`https://training.github.com/kit/downloads/github-git-cheat-sheet.pdf`

4. Workflow

It can take a little while to develop your own workflow. This is an example of how you can start, modify, and share a project. Adding `git` to your workflow can be awkward at first, but most of the time you will only use 3 commands with any frequency. The goal is to spend as little time thinking about `git` as possible!

4.1. **Begin a Project.** Say you have an existing project, or are about to start a new one. These days I turn every project I work on, even papers or documents for teaching, into `git` repos. Most of these I put on `Github` as well! Here's how to do that.

Navigate into the project's directory and immediately type:

```
$ git init
Initialized empty Git repository in /Users/geoff/Projects/gitdemo/.git/
```

You now have a git repo in your project. ALL `git` data for a repo is stored in `.git/`, located in the top level directory of your project. If you ever want to delete your entire repo and all its history while leaving your project data in its current state, simply `rm -rf .git/*` to remove `git` entirely from your project.

Add some files to your repo. Always have a `README` with some basic information and a `.gitignore` to make git ignore temporary or compiled files. Samples for these files are at the end of this document, feel free to customize/modify them as you need. Once they're ready you can add them to the repo and make your first commit.

```
$ git add README
$ git add .gitignore
$ git add <any_other_files>
$ git commit -a
[master (root-commit) f6e9e0b] First commit.
 2 files changed, 23 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 README
```

When you enter the `git commit` command, a text editor will open with some explanatory text and a place to enter a message your commit. This message was "First commit." You can choose which text editor git uses with `git config --global core.editor vim`, or use the `-m "message"` argument for `git commit`.

Lastly, make a repository on `Github` for your project. Always do this. Give it a name (here `"git-demo"`), a short description, and uncheck the "Initialize with a README" box. Click "Create Repository". Then type the following into your shell (replacing `"username""` with your own username of course!):

```
$ git remote add origin git@github.com:username/git-demo.git
$ git push -u origin master
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 518 bytes | 0 bytes/s, done.
Total 4 (delta 0), reused 0 (delta 0)
To git@github.com:geoffryan/git-demo.git
 * [new branch]      master -> master
Branch master set up to track remote branch master from origin.
```

Refresh the `Github` page and you'll see your repository! Voila, a new project.

4.2. **Just regular working.** This is what you'll be doing most of the time. Modifying files, testing new features. Whenever you have implemented a new feature, patched a bug, or completed some small amount of work, you should make a new commit. You can make as many new commits as you want, some people commit

every few minutes. If your project has code, it is a good idea to check the code still compiles before you commit.

When you are ready to commit run a `git status`.

```
$ git status
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

        modified:  src/func.c

no changes added to commit (use "git add" and/or "git commit -a")
```

You have modified a file, and your local repo is ahead of `origin`. That is fine, you're ready to commit.

```
$ git commit -a
[master 5dcacb3] modified func
 1 file changed, 1 insertion(+), 1 deletion(-)
```

And that's it! You are version controlling your code.

After a little while, at the end of the day or after you've implemented and tested a new feature, it's probably time to push your updated version of the code to `origin`, the remote repo on `Github`. This serves to back up your work, share your work with others on the project, and as a nice milestone for you to say "I accomplished something today."

Do a `git status` to double check all your changes are committed.

```
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
nothing to commit, working directory clean
```

Everything is clean, so you're good to push!

```
$ git push origin
Counting objects: 11, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (11/11), done.
Writing objects: 100% (11/11), 1.11 KiB | 0 bytes/s, done.
Total 11 (delta 2), reused 0 (delta 0)
To git@github.com:geoffryan/git-demo.git
   f6e9e0b..5dcacb3 master -> master
$ git status
On branch master
Your branch is up-to-date with 'origin/master'.
```

```
nothing to commit, working directory clean
```

This assumes no one else has pushed to this branch since your last push, which is usually the case. We'll cover more complicated things later.

A normal workflow for the day looks like: `work` → `status` → `commit` → `work` → `status` → `commit` → `work` → ⋯ → `commit` → `status` → `push`.

5. Sample Files

Some sample files.

5.1. **README.md.** Basic `README` file.

```
# My Project #
My Name

This is my project, it does really neat stuff!

Installation:
Requires MPI and HDF5.
$ cp Makefile.in.template Makefile.in
$ make
```

5.2. **.gitignore.** The `.gitignore` instructs `git` not to track certain files. These may compiled binaries, temporary files, data files, anything that is not completely necessary for the project.

```
#Basic gitignore file

# Compiled binaries
bin/*

# C object files
*.o

# Compiled Python files
*.pyc

# OSX Junk
.DS_Store

# Latex
*.log
*.aux
*.synctex.gz
*.pdf
```