THE RECOMPILER

Menu

a magazine about building better technology, together

SHOP

How to Teach Git

by Georgia Reh

Version control is a necessary piece of the open source community and Git has an unfortunately steep learning curve. It is easy to forget that the tools we use everyday are not actually intuitive for beginners. The purpose of this article is to help educators anticipate where their students' confusion and questions might be coming from. Here is what I have learned from teaching Git to beginners, so you do not have to make the same mistakes.

This article is not an outline or a road map you can follow. This article is the caution sign along the side of the road telling you where there are sharp turns and potholes. There is no good one-size-fits-all curriculum, so instead this is a collection of the problems you might come across while teaching Git and some tips for how to avoid them or deal with them when you get there.

What is Git?

Before you start teaching how to use Git, start by giving your newbie an understanding of the environment Git exists in and the problem that it solves. A lot of the confusion that occurs later on (related to remotes, branching, commit history structure) can be avoided if your beginner understands why Git exists.

When explaining Git, do not say "it's version control". This isn't helpful, especially to beginner programmers. That phrase only helps if they already know some other version

control system.

I like to say that Git is a tool that protects yourself and others from yourself and others. Git exists so you can modify/change/break/improve your code, secure in the knowledge that you can not ruin your work too badly because you created save points along the way.

When teaching Git, try not to get side tracked and accidentally teach graph theory or what a hash function is or any other cool concept underlying Git. It is easy to start talking about the things you think are interesting, but just focus on the part the student needs to know. They will be learning a lot of new information in this process and adding on unnecessary levels of detail will compete for attention with necessary information.

You're teaching your student to drive, not how the engine works.

A lot of people teach technical skills by giving a set of instructions in order and expecting the student will absorb the procedure, the concept, and what to expect all at once. It does not work that way. By teaching this strange set of instructions without meaning or mechanism explanations you're basically saying "slaughter a goat on the full moon in an open field near running water and say some phrases in Latin and everything will be fine, but if you get any part of this wrong (some of which I might have forgotten to tell you about) everything will be irreparably ruined."

You must explain the terms before using them and then use them so that your student can Google when they get confused later. This seems basic to say, but we all fail at this regularly, so explain what the terminology means before using it. When teaching a new tool, try not to use incantations, and instead use analogous phrases or metaphors.

For tips on constructing good metaphors, see "Bad Metaphors" by Thursday Bram in the 0th issue of The Recompiler.

The Local Workflow

Init:

When teaching `git init` actually say the word 'initialize', because 'init' sounds like 'in it', which can be confusing.

Make sure you explain that this will have git keep track of changes in your current directory and if they leave this directory, there is not a way for git to keep track of changes

elsewhere. If your student has a firm grasp on what `ls -la` does, then I recommend showing them that `git init` created the subdirectory .git. You can even show them what happens if you delete .git.
Also, this is a good time to use and explain the words ‘repository’ and ‘repo.’

Staging Area:

The staging area is a confusing concept so this would be the ideal time to break out the whiteboard and start drawing pictures. By using terms like ‘staging area,’ you’re invoking a sense of moving things spatially, so be careful and make sure your student understands that files are not being moved from one directory to another. Moving files from the ‘working directory’ to the ‘staging area’ and then the ‘repository’ does not move them but actually changes how Git is keeping track of them.

Status:

Encourage your student to use this command constantly. Since it won’t change the state of things, you can use it at any time whether state has changed or not to help them see. Teach them to use this a lot because it will be useful in the future when they’re confused. This command should be at the top of their list of “how do I figure out what happened” tools.

Add:

Sometimes students think add means “start tracking this file” rather than “include the changes from this file.” To avoid this confusion, don’t just use the incantations: explain the steps, and walk them through multiple commits so you can add a file which had been included in the previous commit.

`rm` and `mv` can be very confusing to learn alongside add, and I recommend waiting until a later day to teach these tools because they probably won’t need them for a while. Let them get used to how `git add` works and feel comfortable with it before getting to these.

Commit:

More so than many of the other commands, forcing yourself to explain the concept of a commit as much as possible before using the word commit is useful, in part because commit has conflicting meanings in colloquial English. Explain that commit is a save point or a check point. If you’re talking to someone who likes playing video games, this is an

ideal time to use video game metaphors.

You can also talk about a commit as a collection of changes that need to be made together. Especially with beginner programmers, explain this concept with non-code examples, because it might not be clear that saving files separately could leave the code base in a broken state.

Always make sure you teach `git commit -m "message"` because if they forget to use the -m flag then git will open vi and ask them to enter the message and save the file. No one wants that. Using vi without prior experience is a confusing and potentially scary process so teach commit as if commit without -m does nothing.

This is a great time to talk about documentation as well.

A strange confusion I've run into is some beginners think `commit` means `push.` My theory is that a lot of people teach `push` as the command immediately after `commit` and it is a more recognizable concept so they latch on to that. However, learning 'commit' as a separate command, rather than a step that is a part of pushing (or, worse, synonymous), is important. To avoid this problem, walk through the process of adding and committing a few times before setting up and pushing to the remote repository. This will separate `git commit` from `git push` and help show that the commands to set up the remote repo are not a part of the workflow.

Log:

`log`, much like `status`, is an incredibly important tool that doesn't get enough attention early on. Many tutorials do not show how to recover old states, despite the fact that the whole point of Git is to be able to recover old versions of your code. You should teach `git log` on day one, with no exceptions.

Show them how to recover a previous state using the name of the commit, but do not teach them what a hash function is. If they already know about hash functions, feel free to use the terminology, but if they have never heard of a hash function, do not say "hash" and do not try to explain it right now, just say that the random sequence of numbers and letters is the "name" of the commit. They are only going to retain so much information, and we are not teaching hash functions.

On the first day of teaching Git, just tell beginners history is immutable or permanent— —teach Git as if the log will never be altered and they can always get back to anything. Early on, it is more important that beginners feel like they have a safety net than they know that

the safety net can be ripped open. Someday, they will need to know about `rebase`, but for now, keep things simple and don't give them the tools to accidentally delete something important.

Diff:

Much like `log`, this is a function that is not always taught right away despite the fact that it only gives information and doesn't change anything. Teach `diff` on day one. Show them what `git diff` and `git diff commit-hash` do. Being able to see what you changed before committing will help people review and get a clearer understanding of what the commit will keep track of.
I recommend teaching `git diff` as part of the work flow right before or after `git status`, so that they see which files have been changed and what changed in them.

The next two categories are Remote Repos and Branching. You can teach either one first; there are benefits and drawbacks to either order.

Remote Repositories

Before teaching any command related to pushing and pulling, take some time to explain why one would use remote repos: for backups, collaboration, etc. Once again, everything will go more smoothly if they understand why they are doing these incantations, which is especially important now, since you are about to teach the worst most confusing commands.

```
git remote add origin
git push -u origin master
```

Good luck explaining these. They've never seen the word "origin" before, and -u is short for "set upstream", another word they've never seen before. They will have trouble Googling for help if they don't understand the underlying concept.

Teach that these are just names assigned to the place they will push and pull from (this is easier to explain if you've already taught branches). The names are used so you do not have to type out the whole address every time you push and pull.

Teach these terms like environment variables. Make sure you distinguish between setting these up and pushing. The commands are often taught together but make sure they understand that in the future they don't need to use the whole command `git push -u origin master`. A lot of tutorials they will come across online will not explain that they can have multiple remotes, so you should explain that.

Pull and Fetch:

`git pull` and `git fetch` are hard to explain together, especially if you haven't covered branching yet. It is easier to just teach one of them, because they probably will not need to know both for a while. Teach the other one later. It does not matter which one you teach first, pick your favorite.

Unless you are teaching them Git for a specific code base you want the people you are teaching to push to, just use GitHub.
1. because they should probably have a GitHub account, if they want to contribute to open source projects
2. there is so much documentation for GitHub, so it will be easy for them to problem solve on their own in the future
3. it's free

Be careful when teaching with GitHub though, because GitHub wants all use of Git to be in terms of GitHub, so it is easy to slip into a GitHub-centric workflow.
Make sure to teach that Fork and Pull Requests are not built in to Git, they are GitHub tools.

Clone:

When teaching `git clone` for the first time, use HTTP, rather than SSH. Teaching SSH key-gen is a confusing, time consuming tangent. Understanding SSH is taking focus away from Git. SSH is about optimization, and on day one, you need to focus on keeping everything conceptually simple.

Once your student knows Fork and `git clone`, this is an ideal time to show them that there are three ways to start a project: Fork + `git clone`, `git clone`, and starting locally with `git init`.

Branching

When you start talking about branching, set aside the computer, grab a whiteboard or paper and start drawing a hypothetical Git history. Draw the acyclic directed graph and explain that conceptually this is what the history looks like. As you walk them through creating branches and merging, refer back to your diagram and show how it affects the picture.

Once again, do not take this time to teach graph theory. Avoid graph theory terminology, even if it is hard to remember that learning the graph terminology takes a lot of time and focus that we should be directing towards Git.

The built-in tree metaphor that Git uses will help you a lot right now.

Teach that Git automatically calls the main branch “master”. That is just a name for the “good copy”, “the copy others might use so I should make it clean”, “the version of my code that my boss cares about”.

Show them how to make branches, checkout branches, and checkout a branch name that doesn’t exist yet. You can teach your students how to delete branches later: we only need to give them tools to get stuff done, not tools to clean up yet.

Sometimes beginners get confused about branches being subdirectories. I don’t know what metaphors or descriptions lead to this confusion, but hopefully if you have explained the tree structure well and what individual commits actually “know”, you can avoid this confusion.

Merge:

When you teach branches, teach them how to merge and choose an example case that will create a merge conflict.

Learning how to merge branches without resolving merge conflicts is not learning to merge. You need to teach resolving merge conflicts right away, because they will definitely encounter merge conflicts in the wild early on. Git is going to throw a scary error message that says “CONFLICT” in all caps followed by “Automatic merge failed”. If this is not a familiar, it can be intimidating and confusing.

Once you have walked them through making new branches with incompatible histories and a failed merge, explain that Git is asking for help and human input, because although it is very smart, it is still pretty dumb. You need to talk about merge conflicts without using the words merge conflict, even though it sounds self explanatory.

Open up the files in question, and show them how to find the patches of code that generated the error. Git marks these problems the same every single time. You might feel compelled to explain HEAD at this time, but suppress that urge, it probably will not help clarify the situation.

At this time, I recommend explaining that Git will let a user commit if they delete the lines of arrows and equals signs, regardless of what they keep between them. Git does not care if you take pieces from one history and pieces from the other. Git only cares if a human looked at it and said “yeah this looks good”.

Once you have fixed the conflict in the two histories, commit the changes and show your student the logs again so they can see Git treats merging just like regular commits. This is also a great time to talk about writing useful commit messages.

Also, now that your student has seen branching and merging, show the log history of both branches so they know log only shows the history of one branch. Refer back to your drawings of the tree at this time.

If you have already talked about remote repositories, you can teach merge conflicts as a part of pull requests to prepare them to contribute to other projects.

Cheat Sheet

In case you are having trouble figuring out how to explain a command without relying on the words in the command, I’ve provided a few examples:

Local Workflow:

`git init`: put Git in this folder so that it keeps track of changes to files in this folder and subfolders

Working Directory: the directory you’re writing code in

Staging Area: files are in the staging area if the changes in them will be included in the next save point

Repository: everything Git is keeping track of

`git status`: show me which files have been changed and which ones are ready to be committed

`git add filename.txt`: include the changes to this file in the next commit

`git rm filename.txt`: don’t teach this on day one!

`git mv filename.txt otherfile.txt`: don't teach this on day one!

`git commit -m "commit message"`: wrap up all these changes and save them together with a short description of the changes

`git log`: show a history of all commits

`git diff`: show what is different from the last commit line by line

`git diff 234nod`: show what is different between the commit 234nod and current state, line by line

Remote Repository

`git remote add origin address-of-remote`: make address-of-remote a new place to put my code and call it "origin"

`git push -u origin master`: push my code to the location origin points to, on the master branch, and also in the future I will pull code from this same location

Upstream: where I will pull code from in the future

Origin: where I put backups or share my code

`git pull`: grab code from another repository

`git fetch`: grab code from another repository

`git push`: save my history and changes in another location

Fork: I want a GitHub repo that looks like someone else's repo

Pull Request: I made some changes that I would like you to include in your repository, please accept them

`git clone`: give me the code at this location

Branching

`git branch`: what are all my branches? or what are all the names of the different versions of my code?

`git branch feature`: make a new branch/version of my code with the name feature

`git checkout feature`: move to that branch/version of my code so I can make changes to that branch/version of my code

`master`: the name of the branch which should be the official, working, well documented, version of my code

`git merge`: combine the history of two branches so I can have the changes from both in one place

Merge Conflict: Git does not know how to combine two histories and needs human assistance

`git rebase`: don't teach this yet!

`git cherry-pick`: don't teach this yet!

`git bisect`: don't teach this yet!

`git amend`: don't teach this yet!

`git reflog`: dear god, don't teach this yet!

`git blame`: don't teach this yet!

HEAD: don't teach this yet!

Acyclic Directed Graph: do not teach them graph theory right now!!!

`git rebase`: seriously don't teach this on the first day!!?!

A few other tips

If you're teaching a workshop with a lot of people, have them install Git and make a GitHub account ahead of time to save time.

Anticipate Windows users and be prepared for the strange ways in which Git bash works differently.

Partner students up so there are fewer different machines to worry about. Make sure the partners have the same operating system so that they do not learn Git in an environment they will never use again.

If you are teaching an individual, give them a project to work on so they can use Git in context. It is harder to pick a project that would work for a group of people, and it will be hard to monitor all of them, of course. But if you can use a project, especially for teaching a single person, the process is a lot easier.

When pair programming, let your mentee type. No one has ever learned to drive from the passenger seat.

You will inadvertently do the correct thing even if it is not what your student tells you to type the wrong thing. For example if the mentee says “and then we should… commit?”, an experienced git user will automatically type `git commit -m “stuff”` even though the beginner didn’t say the words “git” or “-m”

Don’t forget:

Build a context for your student before teaching commands by explaining the problem Git solves, drawing pictures, and using metaphors.

When you teach the commands, make sure you explain them without using the name of the command in the explanation.

Be choosey about what to teach right now and what they can learn later.

Georgia Reh is a mathematician, science educator, programmer, rock climber, immediate friend of every dog she’s ever met, cheese enthusiast, cocktail researcher, probably addicted to caffeine, freakishly good at minesweeper, and professionally trained marshmallow roaster.

Share this:

Twitter | Facebook 19 | Google | Tumblr | WhatsApp | Pocket | Email

LEARN MORE

JOIN OUR NEWSLETTER

Stay up to date on all the latest news from The Recompiler. We publish a quarterly print and online magazine, biweekly podcast episodes, and ongoing technology news and tutorials.

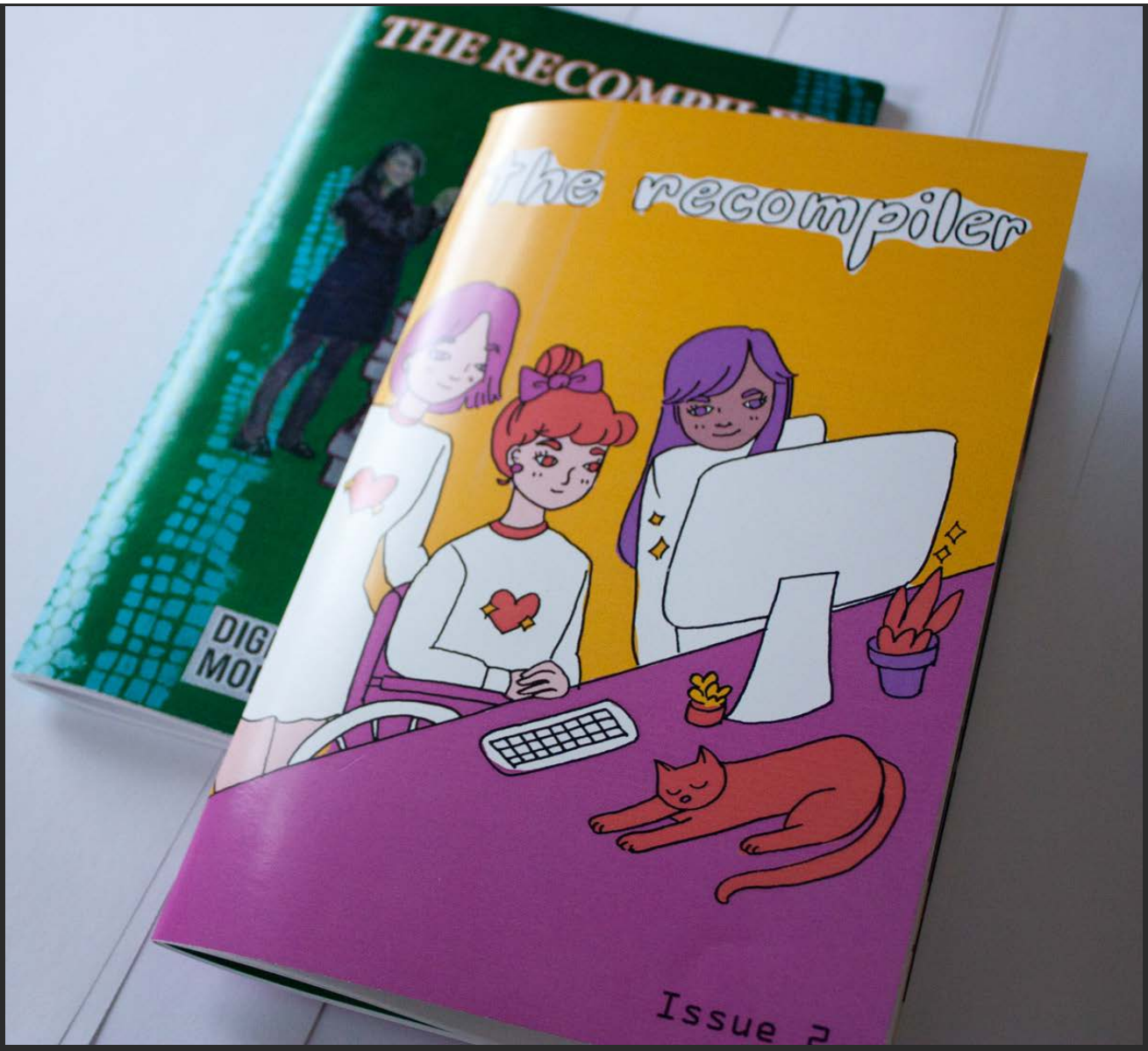
RECENTLY

Looking back on 2016

Episode 23: Congrats, we survived 2016!

Hiring beyond white women and eliminating cultural appropriation

ORDER BACK ISSUES



ABOUT US

The Recompiler is a feminist technology magazine launched in 2015. Our goal is to help people learn about technology in a fun, playful way, and highlight a diverse range of backgrounds and experiences.

LISTEN IN

CONTACT

The Recompiler
P.O. Box 8883
Portland, OR 97207

info@recompilermag.com