Git and GitHub Walkthrough for Common Open-Source Workflows

Ross Tate

September 13, 2019

1 Getting a Repo

Generally there will be an orgranization account (as opposed to individual account) that owns the repo you want to contribute to. For example, my research group has an account called `Tateology`, and in that organization I have created a repo called `cs5152playground` for you to play around with in order to learn git and GitHub. Here is how you get a copy of that repo. Note that this works even if the repo is owned by an individual rather than an organization. (I assume you have created and are logged into your own individual GitHub account, and that you have given me your account so that I can give you access to the repo.)

1.1 Forking a repo

If all you want to do is get the contents of the repo onto your computer, you can skip this step. But if you want to actually contribute changes to the repo, typically major repos are configured so that you are not allowed to directly "push" changes to them from your computer. Instead, you must request that an administrator of the repo "pull" your changes from your *online* repo into the main repo. This means you need to have a version of the repo *on GitHub* from which they can pull the changes. So generally the first thing you do is "fork" the repo you want to change, which prompts GitHub to make a copy of the current state of that repo and associate that copy solely with your individual account.

Go to `https://github.com/Tateology/cs5152playground` and click the "Fork" button in the top-right corner and let GitHub do its thing. Now there will be a repo at `https://github.com/YourUsername/cs5152playground`.

1.2 Cloning a repo

Now you have an online repo that you want to get onto your physical computer. For this, you need to "clone" the repo. If you're using *GitHub Desktop*, then you can do so through the appropriate "Clone a repository from the Internet..." button. If not, execute `git clone https://github.com/YourUsername/cs5152playground`. This will copy all the current contents of your online `cs5152playground` fork into a (newly created) subdirectory named `cs5152playground`.[1]

Make sure that you use `YourUsername` rather than `Tateology` in the URL; both will work, but the latter will configure the repo to push its changes to `Tateology`'s repo rather than your own online fork, and `Tateology`'s repo has been configured to disallow such direct pushes. Of course, this can all be reconfigured after the fact.

If you feel you screwed up, you can just delete the newly created folder and try again. Deleting the folder won't affect anything anywhere else.

Move into the new folder when you're happy with your clone.

1.3 What comes with a clone?

Note that cloning a repo doesn't just copy all the files in the repo onto your computer. It in fact copies the entirety of the repo, including things like its full history. For example, you can execute `git log` to see the

[1] Use `git clone https://github.com/YourUsername/cs5152playground dir` if you want it go into some (newly created) subdirectory named `dir` rather than `cs5152playground`.

recent history of the repo.

2 Getting Updates

You now have a copy of the repo on your own hard drive! But before we worry about making changes to the repo, let's worry about keeping up to date with changes that others are making to the main repo. git and GitHub don't do any sort of automatic syncing; if you want to get the up-to-date version of the main repo, you have to manually "fetch" or "pull" those changes.

2.1 Simulating a change

Go back to `Tateology`'s repo on GitHub (not your own). Click the "Create New File" button. Generally this isn't how files are added to a repo, but GitHub provides it as a convenience feature, so we'll use it to simulate someone making a change. Name the file `YourName.md` (where `md` stands for Markdown, a language for easily writing nicely displayed text files). Type whatever you want into the body of the file.

Scroll down to the "Commit new file" section. You'll be "committing" a change that adds this new file to the repo, so give the commit a short description of what it's doing, e.g. "Create YourName.md". Notice that this short description is in the imperative mood. That is, it's "Do something", rather than "Something was done", "Did something", or "Your Name did something". This is the convention for describing commits. You can also give a longer, more detailed description in the box below, one that might include references to issues that are being addressed by the commit or what not, but ignore that for now.

Finally, click the "Commit new file" button. You've now directly changed the main repo! Again, this isn't normally how changes are made, but it's an easy way to simulate someone making a change to the repo that you want to pull.

Now if you go to your own fork of the repo or to your computer, you'll notice that the new file isn't there. Even though git and GitHub know that you're the one who made the change, they don't automatically propagate that change anywhere beyond the repo you made it in.

2.2 Adding the `upstream` remote repo

You might sensibly think that the way you'd get that new change onto your computer is to first press some button somewhere on GitHub to sync it to your own fork online and then pull that change from your own fork onto your physical clone, but you'd be wrong. In fact, you have to fetch that change onto your computer directly from the main repo and then push the change to your own fork yourself. But right now your physical clone has no convenient access to the main repo, just to your fork!

To fix this, you have to register a new "remote" with your physical clone, unless you used *GitHub Desktop* to clone the repo, in which case this step may have automatically been done for you. Execute `git remote -v` and you'll see `origin` `https://github.com/YourUsername/cs5152playground` listed twice: once with `(fetch)` and once with `(push)`. `origin` is the conventional name for the remote repo that is your own online fork. Its configuration comes from the fact that that's the repo you cloned. But notice that there's no mention of `Tateology`'s repo anywhere.

To fix this, execute `git remote add upstream` `https://github.com/Tateology/cs5152playground`. Now if you execute `git remote -v` again, you'll also see Tateology's repo listed twice, but with name `upstream`. `upstream` is the conventional name for the remote repo that is the real home of the project.

2.3 Fetching changes from `upstream`

Now your clone has a pointer named `upstream` pointing to the main online repo for the project. To update your repo with all the changes in that project, execute `git fetch upstream`.

Git will say a bunch of stuff indicating that it has downloaded various "objects" from the remote repo. But if you look at your directory you still won't see your new file in there. Why is that?

Recall that I said that a clone isn't just a copy of all the files of a repo. Rather, it is a copy of the repo itself. One thing a repo is comprised of is a forest of commits. One of these commits is your new file. The first thing `fetch` does is copy over new commits that have been added to the remote repo.[2] However, it doesn't actually change your local files to use those commits; it just makes them available so that you *can* use them.

2.4 Merging changes from `upstream`

So how do we get git to actually change the local files? Well the second thing `fetch` does is update some local pointers into this forest of commits. In particular, it updates the "remote branches" associated with `upstream`. Execute `git branch -r -v`, and you'll see in the resulting list an entry of the form `upstream/master <alphanum> <commit-message>`. The commit message should be the message you provided when you committed your file, unless someone else made another change between the time you made your commit and executed the fetch. Regardless, the short alphanumeric sequence in that entry is the pointer to the commit that was the "head" of the "master" branch of the upstream remote at the time of your `fetch`, and that commit will contain at least your new file. That alphanumeric sequence is called the SHA (pronounced "shaw") of the commit, and in fact it is an abbreviated prefix of the full 40-character SHA of the commit—a prefix that no other commit in the repo currently happens to share. You can directly reference this (abbreviated) SHA yourself if ever you want, rather than indirectly referencing it through a named pointer like a branch.

Now if you execute `git branch -a -v`, you'll see that the (local) branch `master` is currently pointing to a different commit than `upstream/master` was, illustrating that the local `master` (which happens to be the current "branch" of your local files) is out of sync with `upstream`'s `master`. To fix this, execute `git merge upstream/master`. What `merge` does is update your current local branch and files to incorporate the changes in the referenced commit. It's useful to understand what "incorporate the changes in the referenced commit" means in more detail.

First of all, you shouldn't think of a commit as a collection of *changes*; rather a commit should be thought of as an entire snapshot of the file system *along with* a (non-linear) history of past commits. So the first thing `merge` does is look through the histories of the current local commit and the commit to merge to see when they were last in common. Then it goes forward through time and incorporates any changes that were made in either commit since that last common point in time. It tries do so automatically, but sometimes one commit might change a line in a file one way and the other commit might change that same line in another way, in which case `merge` doesn't know how to incorporate both changes into the same line. If this happens, it'll make you resolve the conflict yourself. If not, it'll make a new commit that is the snapshot of how it automatically combined the two commits. It'll then update the current local branch to point to this new commit.

Of course, in this particular case, the commit you're merging in is a direct future of the current local branch. In this case, all `merge` does is update the local files to reflect the contents of the commit being merged in, and then updates the local branch to point to the commit that was merged in. In particular, no new commit is made. This is known as fast-forwarding.

2.5 Getting changes into your GitHub fork

Now if you look in your directory, you'll see your new file! But take a look at your GitHub fork online. Even if you reload, you still won't see your new file there. In fact, you won't see any changes there. You have only pulled those changes from `upstream` onto your local clone, which has no effect on your GitHub fork.

Execute `git status`. It first tells you that you are "on branch `master`", indicating that your current local branch is `master`—more on that later. It then tells you that "your branch is ahead of '`origin/master`' by (some number) of commit(s)." This is saying that you have local changes that are not on your GitHub fork, a.k.a. `origin`. Note that it does *not* tell you that your branch is in sync with `upstream/master`. This is because your current branch was configured by `git clone` to conceptually connect to `origin/master` by default—more on that later.

To get these changes onto your GitHub fork, you need to "push" your local commit to your GitHub fork. Do so by executing `git push origin master`, which indicates to push the current commit (and its history)

[2]Technically, it copies over just the commits (and their histories) that are necessary to make the particular branch pointers that were fetched be valid.

of the local `master` branch to the `master` branch of the `origin` remote repo. Git will do its thing, and now if you look at your GitHub fork online you'll see your file there!

2.6 Streamlining the process

This is called a "triangular workflow" and is very common in the open-source community. You will be doing this a lot, so let's streamline the process a bit.

First of all, you can execute `git pull upstream master` to fetch the current commit for `upstream`'s `master` branch and merge it into the current local branch. This is almost like executing `git fetch upstream` followed by `git merge upstream/master` except that the only remote branch it updates is `upstream/master`.

Secondly, recall that executing `git status` told you how out of sync your local `master` branch was with `origin/master`. But, since you have full control over `origin`, you often care more about how out of sync your local `master` is with `upstream`'s `master` rather than `origin`'s. You actually can configure which remote branch corresponds to a given local branch, unfortunately called its "upstream branch". By default, when you create a local branch it has no upstream branch, but `clone` sets up specifically `master` to correspond to the `master` branch of the repo that was cloned. To change or initialize the current local branch to use, say, `upstream`'s `master` branch, execute `git branch -u upstream/master`. Note that this only reconfigures the *current* local branch, so it won't affect any other branches you might have—more on that later. Now execute `git status` and you'll see that it tells you how out of sync your current local branch is with `upstream/master`.

The other effect of this change is that whenever you do `git pull` without any other arguments while the current branch has an upstream branch, it defaults to `git pull <upstream-remote> <upstream-branch>`.

That covers pulling, now for pushing. Unfortunately, this change also makes it so that `git push` without any other arguments also pushes to that remote branch. But even if you're allowed to push directly to `upstream`, you typically don't want to, at least not by default. Fortunately this can be changed. First, execute `git config remote.pushdefault origin` to change it so that, whenever a repo is not specified for `git push`, it defaults to `origin`. Second, execute `git config push.default current` to change it so that, whenever a branch is not specified for `git push`, it defaults to the *name* of the current local branch. Altogether, this means that `git push` will now push to `origin`'s branch corresponding (by name) to the current local branch *regardless* of whether the current local branch has an upstream branch.

Lastly, if you want to see how out of sync your current local branch is with its counterpart (by name) on your GitHub fork, execute `git log {push}..` (with the two periods). This will list all the commits that are in the current local branch but not in the branch that you would currently push to by default.

3 Proposing Changes

Now that we've learned how to fetch and propagate changes from the main repo, let's go over how to make changes locally and propose them to the main repo.

3.1 Making a branch

Generally, you don't actually want to make direct changes to files while in the `master` branch. The `master` branch is typically reserved for stable stuff and not used for work in progress. Instead, you generally want to make a new branch and do your work in there.

Execute `git branch chutes`, which creates a new branch called `chutes`. Now execute `git branch`, to get a list of your local branches. You'll see `chutes` and `master`, and there will be a `*` to the left of `master`, indicating that `master` is the current local branch. Thus, all `git branch chutes` did was create a new branch; it didn't change which branch is the current local branch.

Before we learn how to switch between local branches, let's look at the current branches a little closer. Execute `git branch -v` and you'll get that same list, but now it also tells you the SHA and short description of the "head" commit of each branch. Note that they're the same, because when you execute `git branch chutes`, by default it makes the head of the new branch the same as the head of the current branch.

But `git branch chutes` doesn't copy all aspects of the current branch to the new branch. To see an example, execute `git branch -vv` (where the double `v` is intentional). This provides that same list *and* indicates the upstream branch for each branch. Note that `master`'s upstream branch is `upstream/master`, as we set up before, but that `chutes` has no upstream branch. This means that executing `git pull` without any additional arguments while `chutes` is the current branch won't work because it won't know where to pull from.

We'll fix that manually later, but for now let's make yet another new branch (keeping around `chutes` for later) in such a way that it is configured to pull from `upstream/master`. Execute `git branch ladders upstream/master`, which creates a new branch called `ladders` whose head is whatever `upstream/master` is, *and*, because `upstream/master` is a remote branch, configures the new branch so that `upstream/master` is its upstream branch.

But the current branch is still `master`. To change the current branch to `ladders`, execute `git checkout ladders`. Note that this sequence of creating a new branch and then checking it out is very common, so as a convenience you could have used `git checkout -b ladders upstream/master` as a shorthand for `git branch ladders upstream/master` followed by `git checkout ladders`. Before moving on, you might want to execute `git branch -vv` to check that everything is configured as you expect, i.e. `ladders` is the current branch and has `upstream/master` as its upstream branch.

3.2 Making a commit

Now add a line to `YourName.md` with the text "I like ladders." and save the file. Then execute `git status`. It'll tell you that there are "changes not staged for commit" and then indicate that `YourName.md` has been modified. Understanding this properly takes a bit of explaining, but it's well worth understanding properly, especially when you have to start doing more complex things like `git reset`.

On your computer, there are conceptually three "trees" that `git` keeps track of, in addition to branches (and stashes) and such. One tree is known as the "working directory". This tree is the actual files on your computer. Another tree is known as the "`HEAD`". This tree is the head commit of the current branch. And, conceptually in the middle, there is another tree known as the "Index". This tree is the snapshot you currently are proposing for your next commit, and as such it is also referred to as the "staging area". This means there can be a difference between changes you have made in your working directory, i.e. changes you have made to the actual files on your computer, and changes that you have specified should be incorporated into your next commit, i.e. the next snapshot you want to add to the repo to share with others.

Going back to the output of `git status`, it is saying that there are changes in your working directory that are not in your Index, i.e. staging area. In other words, there are changes in your working directory that have not been "staged". Since the goal is to make a change that will eventually be incorporated into the master repo, you need to stage the change to your file.

To do so, execute `git add YourName.md`. This "adds" *all* changes you have made to `YourName.md` to the staging area. Note that sometimes you only want to add some of the changes in a given file to the staging area. For this you can execute `git add -p YourName.md`, i.e. `git add --patch YourName.md`, and interactively select which "hunks" of changes you want to add, or you can use *GitHub Desktop* to select lines with a more discoverable interface.

Now if you execute `git status`, it'll say "changes to be committed" rather than "changes not staged for commit". So let's commit those staged changes by executing `git commit -m "Add ladder preference to YourName.md"`. This makes a new commit comprised of the contents of Index, gives that commit the descriptor in the quotation marks following `-m`, and changes the `HEAD` of the current branch to point to the new commit. Alternatively, you could just execute `git commit`, in which case it'll pop up a text editor and have you provide a descriptor. You must provide a descriptor; if you leave it blank, git will abort the commit. This is because descriptors are very useful for looking back through history. See this article for advice and motivation on writing good descriptors: `https://chris.beams.io/posts/git-commit/`. For now, use `-m` when all you need is the short descriptor, or elide `-m` when you want to provide a more detailed descriptor, but then still provide a short descriptor as the first line, followed by a blank line, followed by the detailed descriptor.

Now execute `git status` and it will tell you that you are 1 commit ahead of `upstream/master` (the upstream branch of the current branch `ladders`). If others have made changes to `upstream/master` in the meanwhile, it will also tell you that you are behind so many commits, but don't worry about that right now. As we'll see, you don't always need to be completely up to date, and constantly updating unnecessarily can create a bunch of "merge" commits that clog up the repo history.

Next execute `git log`, and the first thing it will list is your new commit because that is now the `HEAD` of the current branch.

3.3 Uploading a commit

Now you've made a commit whose snapshot contains your file, but that commit is sitting on your computer and nowhere else. Typically, no one has remote access to your computer, which means no one has a way of getting this new commit at the moment. You need to share it.

You could try executing `git push upstream ladders:master` to push from the local `ladders` branch to the main repo's `master` branch to share your commit, but you will get an error because, as with most open-source projects, `Tateology/cs5152playground` is configured to disallow non-administrators from directly pushing to it. So to make a change to the main repo, you must instead get an administrator of the main repo to pull your changes from you. For that to happen, you must get those changes somewhere they can pull them from, which typically does not include your computer. That is why you have your own fork of the repo on GitHub.

Remember that we used `git config` to make it so that `git push` by default pushes to the `origin` remote repo, i.e. your fork on GitHub, and specifically to the branch on `origin` with the same name as the current branch, i.e. the branch you currently have checked out on your computer. Right now there is no branch named `ladders` on your GitHub fork, but that's okay; `git push` by default will just create one for you. So simply execute `git push`, or `git push origin ladders` if you didn't choose to change the default configurations for `push`, and your changes will now be available on your GitHub fork.

To witness this fact, go to `https://github.com/YourUsername/cs5152playground`. If you try to look at `YourName.md` right now, you won't see your changes. That's because you pushed your changes to the `ladders` branch, and currently you're looking at the `master` branch—more on that later. On the main page for your fork, i.e. `https://github.com/YourUsername/cs5152playground`, find the dropdown that says "Branch: **master**", and change it to `ladders`. Now if you look at `YourName.md` on your fork, you'll see the new line of text you added.

3.4 Making a pull request

Go back to the main page for your fork. You should see a yellowish box mentioning `ladders` and containing a green button labeled "Compare & pull request". Click that button!

This'll bring you to a new page. Near the top there will be four dropdown menus with a leftwards-pointing arrow in the middle, i.e. `<target-repo> <target-branch>` ⇐ `<source-repo> <source-branch>`. These dropbowns indicate that you are in the process of creating a request to pull the contents of the `<source-branch>` of the `<source-repo>` on GitHub into the `<target-branch>` of the `<target-repo>` on GitHub. That is, you're conceptually asking an administrator of the `<target-repo>` to execute `git pull <source-repo> <source-branch>` while the `<target-branch>` is currently checked out.

Currently the left boxes should specify the `master` branch of `Tateology/cs5152playground`, and the right boxes should specify the `ladders` branch of `YourUsername/cs5152playground`. If you click on the rightmost box, you'll see that you could alternatively select `master`, but you'll also notice that `chutes` isn't there. This is because although you created a `chutes` branch on your local machine, you never pushed that branch to your GitHub fork.

Scrolling down to the bottom, you'll see a summary of the changes that accepting your pull request would cause at the moment. This summary includes the commits that are currently in the branch to be pulled from that are not currently in the branch to be pulled into. It also includes diffs of the files that would be changed by incorporating these commits.

Scrolling to the middle, you'll see a field where you specify the title of your pull request. By default this is generated from the description of the most recent commit in the pull request. Often that default isn't what you want because usually pull requests are larger scale, so you'll want to change the title to summarize the high-level purpose of the pull request. But for this case, the default title is fine. Note, though, that this title can't be changed once the pull request is created!

Below that field you can provide a more detailed description (in Markdown) of the pull request. One particularly common thing to do here is reference GitHub issues that the pull request is addressing, but we won't get into that here. For now, just leave that blank.

Below that there is a checkbox labeled "Allow edits from maintainers." This makes it possible for others to modify your pull request. Sometimes this is useful because the administrators of the target repo can facilitate the process; other times this is bad because your pull request can become out of sync with your local clone(s). We'll discuss this more with your next pull request, but for now unclick that box, which makes it so that only you can revise your pull request.

Finally, click the green "Create pull request" button. You've just *proposed* your first change to the main repo! But it's up to them to decide if they actually want your proposed change. Often they do, but very often they first want you to revise your proposal. We'll simulate that process next.

4 Revising Proposals

After you clicked the "Create pull request" button, you were taken to a new page—the page of the pull request you just created. Notice that this page is in `Tateology`'s repo, not yours. This is because the administrators of `Tateology` want to easily manage what pull requests are being asked of them. Also, if the pull request is accepted, it's `Tateology`'s repo that gets changed, not yours.

4.1 Reviewing the proposed changes

You actually have been given write access to `Tateology`'s `cs5152playground` repo. That means you can accept, i.e. merge, the pull request right now if you want to. But don't do that. Instead pretend you have changed roles and you are now an administrator who wants to make sure the contents of `Tateology`'s repo are in good order.

After reading the description to find out what the pull request is about, one of the first things you'll want to do is actually look at the changes the pull request would make to the files. To see this, click the "Files changed" tab.

This tab presents you with the changes the pull request would make to the files. In particular, you'll see that the pull request would add the line "I like ladders." to the `YourName.md` file.

Let's suppose you're fine with the intent of the change, but maybe you want more clarity. After all, everybody likes ladders. They're super useful! So let's ask the contributor to clarify their documentation.

Hover your mouse over the new line of text in the diff and you'll see a blue plus sign appear to the left of the line. Click it, making an interface appear that lets you comment that line in the pull request. This comment will have no effect on the actual files, but anyone looking through the diff in the pull request will see your comment, and anyone reading through the "Conversation" tab will see your comment along with a snippet around the line you commented on. Plus once you make the comment, presumably the people involved in the pull request will be notified. Make a comment asking the contributor to clarify what it is they like about ladders; for simplicity, do so by clicking "Add single comment" rather than "Start a review" after you've composed your text.

4.2 Revising your pull request

Now switch back to the role of the contributor requesting the change. Let's go into how to make the requested clarification. On your computer, you should still have the `ladders` branch checked out. Reopen `YourName.md` and change that line of text to "I like to play on ladders." and save. Stage the change, i.e. execute `git add YourName.md`, commit the change, i.e. execute `git commit -m "Clarify why Your Name likes ladders"`, and push the new commit to the `ladders` branch of your GitHub fork, i.e. execute `git push` or `git push origin ladders`.

Now go to the page for your pull request and, if necessary, refresh. You'll see your new commit appear in the "Conversation" tab and the "Commits" tab. This is because a pull requests asks to pull from a particular *branch*, not from a particular *commit*, and so updating the contents of that branch on your fork effectively updates the pull request—yet another reason to use branches!

Next if you look at the "Files changed" tab, you'll see that the diff is updated to reflect the changed line. Notice, though, that the diff doesn't mention that your last commit effectively deleted the old version of that line and inserted a new one; it just shows the end result of all the commits together. Also notice that the comment is gone from the diff, since it was associated with a line that was deleted. You can still see that comment in the conversation, though, and in the diff for the original commit if you click on it in the "Commits" tab.

4.3 Reviewing the proposed commits

Switch roles to administrator again and suppose you are reviewing the revised pull request. The changes to the files look good now, but there's more to a repo than just its files. You also care about the history of the commits, since one of the points of version control is to be able to look back through the history of the repo and do things like identify the point in time where a new error was introduced.

If you go to the "Commits" tab, you'll see that there are two commits. But this pull request is just adding a single line of text. The older commit just shows an intermediate point that no one cares about anymore. As such, it'll simply muck up the history.

Go to the "Discussion" tab, scroll down to the bottom, and make the comment "Please squash your commits."

4.4 Squashing commits

Now switch back to the role of contributor. Squash? What does squash mean? It means to compress your commits together into one commit comprised of all their changes (or to at least compress away the insignificant commits, leaving only a few commits marking the major steps).

There are two ways to do this, each with their own pros and cons. Right now we'll go over the "safer" method, which is to "rebase" your branch.

Generally speaking, rebasing involves revising your branch to pretend that history played out slightly differently. In this case, we want to pretend that the changes in those two commits were made at the same time as part of one commit. Since we happen to know precisely how far back in time we want to go, we *could* do this by executing `git rebase -i HEAD~2`, which indicates to rebase the commits made since `HEAD~2`, which is notation that calculates the commit that is 2 steps back in time from the `HEAD` of the current branch. However, since our current branch has an upstream branch, we can also do this by simply executing `git rebase -i`. This will automatically figure out the most recent commit in the history that the current branch and its upstream branch have in common, and then it will let you rebase all the commits made since then on top of the contents of the upstream branch. This is very useful for pull requests, presuming that you've set the upstream branch of the current branch to be whatever branch you're requesting to pull into, because it will have you rebase precisely the commits that appear in your pull request.

Regardless of which command you executed, a text editor will appear prompting you to describe how to rebase those commits. If you close that file right away, nothing will happen; the default contents of that file will leave things as is. But we in fact do want to make a change.

Note that the file lists the commits with the oldest commit at the top and the most recent commit at the bottom. After you edit and close the file, git will walk down this list and execute the commands you specified in order.

In this case, you want the rebase to use the *file* changes of both of your commits but to elide one of the actual commits from the *history*. The commands `squash` and `fixup` both incorporate a commit's changes without incorporating it into the history, but they do so by modifying the *previous* commit that was `pick`ed rather than the next commit that gets `pick`ed. So that means in this case you want to at least `pick` the first (i.e. topmost) commit and `squash` or `fixup` the second commit.

The difference between `squash` and `fixup` is small but important. With `squash`, the description of the commit being `squash`ed is appended to the `pick`ed commit's description. With `fixup`, the description of the commit being `fixup`ed is simply ignored. Since we no longer care about the description of the second commit, change the rebase to `fixup` the second commit.

While we're at it, let's improve the description of the compressed commit we're creating. The description of our first commit is fairly low level, so let's make it a little more high level. Rather than `pick` the first commit, change the rebase to `reword` the first commit. For all other purposes, these two commands do the same thing, but `reword` will also prompt you to change the description of the commit.

Now save and close the prompt. Because you used `reword`, you'll be prompted to give a new description for the commit you chose to reword. Change that description to `Add Your Name's preference for ladders`, then save and close the prompt.

Now execute `git log` and you'll see that you've "changed history". That is, not only will your new compressed commit with its new message be listed first, but also the old commits will no longer be around. You're considered to have "changed history" if ever you make a branch point to something that is not a future commit of what it pointed to before, where future is in the sense of the commit histories and not in the sense of physical time.

4.5 Changing history remotely

Now that you've squashed your commits locally on your computer, you need to change your pull request to use the squashed commit. Try to execute `git push`; you'll get an error. The problem is that, because you used `rebase` to change history, your squashed commit is not considered to be a future of the current contents of your fork's `ladders` branch. In order to avoid accidentally losing changes, `git push` doesn't let you push anything that isn't considered to be a future of what's there currently.

But in this case, the whole point is to change history so that the unnecessary intermediate commit isn't mucking up the log. So execute `git push -f`, i.e. `git push --force`, to force the change despite the fact that it changes history. But beware: if someone else had changed that branch in the meanwhile, their changes would be lost by this and no one would even know (for a while). In this case, this is your own personal GitHub repo, so you know this isn't a problem. But in general people are very hesitant to force push to the main repo, which is why any desirable changes to history are strongly preferred to be done before the pull request is merged.

4.6 Merging a pull request

Now if you go to the page for your pull request (and refresh), you'll see that the "Discussion" has been updated to indicate that you force-pushed a change to the pull request. If you go to the "Commits" tab, you'll see that there's just one commit there, and it has your revised description. And if you go the "Files changed" tab, you'll see that the overall changes to the files are the same.

Thus, if you change roles to the administrator reviewing the pull request, everything looks good! So click the "Merge pull request" button. You'll be prompted to give a short description and a long description for the commit that will be made to merge the pull request (even if no one else made changes in the meanwhile). Different teams have different conventions for what to do here, so ask your mentors what they would like if ever they have you be the one to merge the pull request. Just go with the defaults for now and click "Confirm merge".

Once you complete the merge, go to `https://github.com/Tateology/cs5152playground` and click `YourName.md` and you'll see your change!

4.7 Cleaning up branches

Now to clean up. The branches you made are no longer necessary; they've served their purpose. So let's delete them before you forget why they're there.

On your computer, first execute `git checkout master` so that your current branch is no longer the branch you want to delete. While you're at it, reopen `YourName.md` on your computer. Your change to the file is gone! That's because, although `Tateology`'s `master` branch has been updated with the change, you haven't yet pulled that change to your local `master` branch. And when you check out a branch, by default it changes the contents of your files to reflect the contents of the branch you checked out (with important caveats regarding uncommitted changes, which we won't go into here).

Although you could pull those changes now, hold off on that to illustrate a point. Execute `git branch -d ladders` to try to delete your local `ladders` branch. It doesn't work! The reason is that you have a commit in that branch that isn't in its upstream branch (or, if it has no upstream branch, in the currently checked out branch), so git is preventing you from accidentally losing work. Wait, that doesn't seem right? Didn't we just pull those changes in? Well, yes, but you never fetched the update, so the clone on your computer still things `upstream/master` is pointing to the commit before you modified the fill. So execute `git fetch` to get those changes and let git know that they'll be preserved. Now execute `git branch -d ladders` again; the local branch will successfully be deleted.[3]

But this only deletes the local `ladders` branch; the `ladders` branch is still there on your GitHub fork. If you go to the page for the (now merged) pull request, you'll see a "Delete branch" button to delete the branch from your GitHub fork. But for educational purposes, rather than press that button let's do it from the command line. Simply execute `git push origin --delete ladders`, and the branch will be removed from your GitHub fork. And now your no-longer-necessary branches have all been cleaned up! You're all done making your change to the main repo.

Lastly, while you have the `master` branch checked out, execute `git pull` or `git merge upstream/master` so that your change is actually in your local `master` branch, and then execute `git push` so that it's in your GitHub fork's `master` branch.

5 Resolving Conflicts

In all of this, you shouldn't have had to deal with any "conflicts", meaning independent simultaneous changes that git was unable to resolve automatically. This is because you only modified the file with your name, and likewise others should have only modified files with their names, and git can automatically merge simultaneous changes made in separate files. But real-world development doesn't always work out so nicely, so let's cause a conflict to resolve.

Remember that `chutes` branch we made and then ignored? In all this time, it was never updated. So it still refers to the original version of `YourName.md`. We'll take advantage of that, pretending that you started to modify `YourName.md` in `chutes` and, in the meanwhile, someone else did the same and even had the change pulled into the main repo.

Check out `chutes`, i.e. execute `git checkout chutes`, but don't do anything else. In particular, don't merge or pull in any updates. Open `YourName.md` and you'll see that your new line of text is once again gone. Add the line of text "I like chutes." (rather than "ladders") and save. Stage the change, make a commit, and push the commit to your fork (specifically the `chutes` branch of your fork, though remember we configured our clone so that that would be the default behavior).

Now go to your fork online and start to make a new pull request from `chutes`. You might notice the warning indicating that your current proposal can't automatically be merged. Ignore that warning; we'll pretend the conflicting pull request was merged in after you opened your pull request. This time, leave the checkbox "Allow edits from maintainers." filled; although we won't technically take advantage of that, down the line we'll pretend we're a maintainer revising the pull request. Click "Create pull request".

5.1 Resolving a conflict

Okay, suppose you now notice that your pull request can't be merged because "this branch has conflicts that must be resolved". Technically, the branch doesn't have any conflicts within itself—rather the branch has changes that conflict with the changes that have been made to the target branch. Regardless, you need to "resolve" the conflict, meaning you need to manually tell git how to combine the two changes.

Note that you don't always want to do this immediately because the main repo will continue to change as you revise your pull request. Often resolving conflicts is done as a last step when everyone's decided the pull request is otherwise good to go. But even if that's the plan, it's good to check what the conflict is once in a while to gauge whether it's something that should really be addressed earlier rather than later.

[3] If ever you want to delete a local branch despite it having commits not in the corresponding remote branch, say because the local branch was experimental and the experiment didn't pan out, you can use `git branch -D ladders` (with a capital `D`) to force the local branch to be deleted.

First, pull in the conflicting changes. Remember we didn't configure `chutes` to have a remote branch, so you'll have to execute the full `git pull upstream master` command.

Notice that it tells you that "automatic merge failed". Execute `git status`. It'll mention that "you have unmerged paths", and indicate that `YourName.md` has (unstaged) modifications. So something happened, but what?

Open `YourName.md`. There will be a line with `<<<<<<< HEAD`, followed by a line of text, followed by `=======`, followed by another line of text, followed by `>>>>>>>` and a SHA. The lines between `<<<<<<<` and `=======` are what you had in your local branch, and the lines between `=======` and `>>>>>>>` are what was in the changes you just pulled in. This is the conflict you need to resolve.

Now if you had just wanted to see the conflict in order to gauge whether it should be resolved now or later, you can execute `git merge --abort` and everything will go back to the way it was before. Remember this command! There will be many times where you will pull and your reaction will be something like "Ah! I wasn't expecting this! I'm not prepared for this! What do I do?!" The answer to your panicked question is `git merge --abort`.

But we were expecting this, so let's resolve the conflict. Do so by replacing the `<<<<<<<` and `>>>>>>>` lines and everything in between with what you believe should be there after the merge. To figure out what that should be, you probably want to research why the change was made to begin with. So copy the SHA after `>>>>>>>` and execute `git log <sha>`. You'll see the most recent commits that led to that change. In particular, there should be a mention of a pull request, and you can go on GitHub to look at that pull request and all of its discussion.

After reading the pull request, you might surmise that the appropriate resolution is to replace the texts with "I like to play chutes and ladders.". Do so and save the file.

Execute `git status` and you'll see that the status hasn't changed. You still have (unstaged) modifications to `YourName.md`. So stage them by executing the familiar `git add YourName.md`. Execute `git status` again and it will tell you "all conflicts fixed but you are still merging."

To finish the merge, execute `git merge --continue`. It'll prompt you to provide a description. The default is rather long, so shorten it to "Merge 'Tateology/master' into 'YourUsername/chutes'" and then save and close the prompt. Now execute `git log` and you'll see your merging commit, along with your commits from `chutes`, along with the recent commits that were pulled in.

Lastly, execute `git push` and then look at your pull request on GitHub and you'll see the merge commit. GitHub should now say that your pull request can be merged, but don't merge it yet!

5.2 Fetching someone else's pull request

Now suppose you're an administrator reviewing this updated pull request. The diffs look good, but you look at the commits and you see that the last commit was simply a merge from the main repo. When you merge this pull request in, then, the history will have a merge of the pull request into the main repo preceded by a merge of the main repo into the pull request. That back and forth seems pointless and will muck up the history.

Normally you'd ask the contributor to fix the history, but suppose you're in a rush and you decide to do it yourself. Fixing pull requests just before merging is generally fine, but this particular fix that you're going to do is considered to be bad form. The reason is that you're going to be changing history, and you don't know what other branches the contributor has made off of this pull request, so your change to history will essentially cause those other branches to be on an entirely separate and hard-to-merge timeline. For this reason, you should only change history for branches that you have control over and that you have full knowledge of how it's been branched off of.

But for the sake of this walkthrough, there will be times where you want to fix someone else's pull request, and there will be times where you want to change history to avoid this back-and-forth merging, so I've combined the two into one lesson.

First, on your machine, check out `master` so that you don't accidentally change the contents of the local `chutes` branch, which is supposedly on someone else's machine. Next, execute `git branch -r -v` just to review the list of remote branches your clone knows about. This list is not necessarily all of the branches that exist in your registered remote repos. They are the ones that `fetch` (and `pull`) have retrieved, but by default `fetch` (and `pull`) are configured to only retrieve certain sorts of remote branches. In particular, your `upstream` repo actually has remote branches for each of its pull requests. This includes all the closed pull requests, so this list of remote branches can get quite long, which is why they are not fetched by default.

To fetch pull request `#<pr>`, execute `git fetch upstream pull/<pr>/head:pr-<pr>`. This fetches the commits referred to by the remote branch `pull/<pr>/head` and then makes the (new) local branch `pr-<pr>` point to whichever commit that remote branch is pointing to. In general, when you execute `git fetch <remote-repo> <remote-branch>:<local-branch>`, it grabs the contents of `<remote-repo>`'s `<remote-branch>` and makes the `<local-branch>` on your computer point to the appropriate commit. So if you wanted to update just your local `master` branch without checking it out, you could execute `git fetch upstream master:master`.[4]

Now you have the contents of the pull request in the `pr-<pr>` branch on your computer, so just check out that branch to look at those contents.

5.3 Referring to the past

We are going to fix the pull request by going back in time and redoing the merge the way we wish it had been done. After revising the merge, we will need to resolve the conflicts. Rather than do so manually, we will reuse the resolutions currently in the pull request.

This means we need to keep the current pull request around. So we will start our fix by checking out a new branch from here. Execute `git checkout -b pr-<pr>-fix`.

Next we need to go back in time in this branch to before the merge was done. The first step to this is referring back in time.

Execute `git show HEAD`. Recall that `HEAD` refers to the current head of the current branch. Because the last commit in the pull request was a merge, there will be a line `Merge: <SHA1> <SHA2>`. Those two SHAs are the two "parents" of the commit that were merged together. We could use these SHAs to refer to the past, but there is a better way.

Typically the notation `HEAD^` refers to *the* parent of the `HEAD` commit. But the current `HEAD` commit has *two* parents, so which one is *the* parent? The mathematical answer is neither, but the conventional answer is the first one. The reason is that when a merge is done, the first parent will refer to what was the local `HEAD` before the merge, and the second parent will refer to the (head) commit that was merged in. Thus typically the first parent of a merge will be the commits that are more relevant to whatever the purpose of the local branch is.

For this reason, `HEAD^` is actually shorthand for `HEAD^1`, which denotes the first parent, as opposed to `HEAD^2`, which denotes the second parent. Similarly, `HEAD~2` is shorthand for `HEAD^^`, which is shorthand for `HEAD^1^1`. For this document I will assume that the first parent is the head of the pull request before the merge was done, and that the second is the head commit that was merged in. But you should execute `git show HEAD^1` and `git show HEAD^2` just to check if something weird happened and you need to swap the numbers as you go forward.

5.4 Resetting to the past

Now that we know that `HEAD^1` refers to the pull request before the undesirable merge was done, we need to revert to that past point. You may have heard of `git revert`, but that is not what we want. `git revert` makes a new commit that changes files in a way that undoes the changes made by an eariler commit, but it still keeps the old commit around. This is useful when you don't want to change history, but completely useless when you are trying to clean up history, as we are now.

The tool we're going to use is `git reset`. But before we go into `git rest`, it is useful to recall the three trees: the working directory (the files on your computer), the Index (the staged changes), and the `HEAD` (the commit that the branch points to). What `git reset` does is change these trees, and different modes of `git reset` change these trees in different ways. Note that `git reset` isn't the only thing that changes these trees: modifying your files in a text editor changes the working directory, `git add` adds changes in the working directory into the Index, and `git commit` makes the Index into a new `HEAD`. So more precisely, `git reset` changes these trees in the reverse direction.

[4]This will fail if it would change the local `master`'s history. If you want it to succeed regardless of changes to history, add a `+` as in `git fetch upstream +master:master`.

You can find a good in-depth overview of what `git reset` does with this trees at `https://git-scm.com/book/en/v2/Git-Tools-Reset-Demystified`. Here we'll go over the most common use case where you specify a degree—`--soft`, `--mixed` (the default), or `--hard`—and a commit to reset to. In the `--soft` case, `HEAD` is changed to the specified commit. In the `--mixed` (default) case, *furthermore* the contents of Index are changed to the contents of the specified commit. In the `--hard` case, *further-furthermore* the contents of the working directory are changed to the contents of the specified commit.

So suppose all three trees are in sync, as they are now in your current branch. Then `--soft` would change the head of the branch but would leave the files and staging area unchanged; thus `git status` would indicate you have a bunch of staged modifications to be committed. Note that this gives you another way to squash your commits that works well in many situations: just `git reset --soft ...` to back when you first started making the changes in your pull request, then `git commit` to put all of those changes into a single commit. Moving on, `--mixed` would change the head of the branch and the staging area but leave the files unchanged; thus `git status` would indicate you have a bunch of modifications that need to be staged. Lastly, `--hard` would change everything to reflect the specified commit, leaving the three trees in sync in a new state. Note that this includes forgetting all modifications you had in progress, regardless of whether they were staged and even including new files you had made, so be careful about using this.

In this case, we want to completely go back in time, so `--hard` is the mode we want to use. Execute `git reset --hard HEAD^1` to reset everything back in time to the state of the pull request before the merge was done. Note that we could have skipped all this by just executing `git checkout -b pr-<pr>-fix HEAD^1` when we first created the branch, but that would have denied us an educational opportunity.

5.5 Rebasing rather than merging

Now we need to incorporate all the changes that have happened in the main repo, or at least the changes that led up to the conflict with the pull request. We could pull from `upstream`'s `master`, but that would just reintroduce the problem we are trying to address. Furthermore, more changes may have been made to the main repo, so it might even introduce new problems. On that note, realize that `pr-<pr>^2` refers to the exact state of `upstream`'s `master` when it was merged into the pull request. So we could execute `git merge pr-<pr>^2` to merge in the exact same changes. Once again, this would reintroduce the problem we are trying to address, but at least it wouldn't introduce any new problems.

Fortunately there is another way to incorporate changes (and history). It has the downside that it changes history locally, but in this case that is okay. Instead of merging, execute `git rebase pr-<pr>^2`. If we think of each local commit as a delta from some starting point rather than a snapshot, this command makes a new timeline in which each of these deltas instead started from `pr-<pr>^2`.

Occasionally, though, it will be unclear how to apply a particular delta for a commit due to some conflicting difference in `pr-<pr>^2`. When this happens, the rebase will pause and ask you to resolve the conflict in much the same manner as a merge conflict. After you resolve the conflict, you then execute `git rebase --continue` to continue applying the remaining deltas. Note that this means you might be asked to resolve conflicts multiple times because subsequent deltas might have further conflicts. On the other hand, `git merge` has you resolve all conflicts at once, which is a major advantage of merging over rebasing. You can also execute `git rebase --abort` if you just want to bail out of the process entirely and go back to the way things were.

5.6 Reusing resolutions

In this case, there is only one local commit, and so only one delta to be applied. But this delta has a conflict for the same reason the pull request had a conflict to begin with. Now we need to resolve this conflict. We could do so ourselves, but remember that the pull request had already resolved this same conflict.

We want to reuse that resolution, but it is inside the merge commit that we do not want to have in our history. Fortunately, git provides a way to grab the contents of a file from a commit without grabbing the commit itself. Execute `git checkout pr-<pr> YourName.md`, which will replace `YourName.md` in the Index and the working directory with the version of `YourName.md` in `pr-<pr>`. So if you look at `YourName.md` you'll see the resolution to the conflict we made before, and if you execute `git status` you'll see that you have one staged modification and all conflicts have been resolved. So lastly execute `git rebase --continue` to move on and finish the rebase.

5.7 Checking your work

Before we push the fix, we should check it. Execute `git show`, which is short for `git show HEAD`, and you will see you've made a commit that replaces the line added by your previous pull request with the new version that effectively combines both pull requests. Furthermore, if you execute `git log`, you will see that your previous pull request is already in the history. This means this new commit won't be considered to be in conflict with your previous pull request because your previous pull request is in this new commit's history.

But let us make sure we actually recreated your new pull request. Execute `git diff HEAD pr-<pr>` to get a comparison of the contents of the *files* (but not the histories) of the current commit with the pull request. You should see nothing, indicating that there is no difference, just as we wanted.

So finally we can push the commit. But we cannot push it to `upstream`'s `pull/<pr>/head` branch because it is a read-only view of the pull request. Thankfully, because the pull request was configured to allow edits from maintainers, we can directly commit to the branch on the fork that the pull request is pulling from. But there's a caveat to keep in mind.

5.8 Safely changing remote history

We are changing history, so we have to force-push our change. However, it is possible that, while making our fix, the pull request has been updated by the original contributor, so our force-push would throw away updates we didn't intent to throw away. Even if we were to check it now, the pull request could get updated between the time we check it and the time our push goes through. This is a classic concurrency problem that we managed to avoid before by knowing that we were the only one modifying that remote branch.

Thankfully, git provides a way to conditionally force-push a change, analogous to compare-and-swap if you are familiar with that operation. To use this, execute `git push --force-with-lease=chutes:pr-<pr> https://github.com/YourUsername/cs5152playgroud pr-<pr>-fix:chutes`, using the URL simply because conceptually this is a one-time push to this contributor's fork (that typically wouldn't be your `origin` remote). The flag `--force-with-lease=chutes:pr-<pr>` indicates to push if and only if the remote repo's `chutes` branch currently points to the head commit of the local clone's `pr-<pr>` branch. The final argument `pr-<pr>-fix:chutes` indicates to push the contents of the local clone's `pr-<pr>-fix` branch to the remote repo's `chutes` branch. If you only said `pr-<pr>-fix`, it would attempt to create a new branch of that name on the remote repo, though that might fail due to lack of permissions. If you just said `chutes`, it would push the contents of the local clone's `chutes` branch to the remote repo's `chutes` branch, which are already coincidentally in sync.

Supposing that worked, which it should in this exercise, you are now happy with the pull request and can merge it in. Don't forget to delete the local `pr-<pr>` and `pr-<pr>-fix` branches you no longer need, using the `-D` flag instead of `-d` to ignore the check that the commits have been merged. And changing roles to the contributor, your pull request has been pulled in, so delete the `chutes` branch on your fork and your clone.

5.9 A problematic alternative solution

An alternative solution to the above fix would be to do the following after checking out `pr-<pr>-fix` as above. First, execute `git reset --soft HEAD^2`. This will change the head of the branch to point to the commit that was merged in, but it will leave the local files and Index unchanged. Thus if you execute `git status`, it will say that there are staged modifications ready to be committed, where those modifications are precisely the changes to the main repo that the pull request would incur. So then you could execute `git commit` and provide an appropriate message. Finally, you could force-with-lease-push the fix to the pull request as above.

This may seem simpler, but it has a *huge* downside: it erases the commit history of the pull request. In particular, even if there were only one commit and you made sure to recreate its message, the new commit would be signed by *you* rather than the original contributor. So you've essentially stolen credit for their work. The more complex solution above, on the other hand, will both preserve the commit history (in a relative sense) and preserve the authors of the commits. If you don't care about preserving the relative commit history, you can manually preserve the author by adding the `--author="Their Name <their-email@their-domain.com>"` flag when you execute `git commit`.

5.10 Preventing the problem

Of course, this could have all been prevented in the first place had the contributor used `git pull --rebase upstream master` to incorporate the conflicting changes from the main repo by rebasing rather than merging. In fact, some developers configure this to be their default. But remember that this changes history locally, which can cause problems when you have other branches off of the current branch already in play, so one has to be very careful to use it as their default.