**What is GitHub? When was it created? Why? By who? What similar platforms exist? Why would you use such a platform? (Answer between 5 and 10 lines)**

Answer:

GitHub is a Web-based Git repository hosting service. It offers all of the distributed revision control and source code management (SCM) functionality of Git as well as adding its own features. Unlike Git, which is strictly a command-line tool, GitHub provides a Web-based graphical interface and desktop as well as mobile integration. It also provides access control and several collaboration features such as bug tracking, feature requests, task management, and wikis for every project.

It was founded on February 8, 2008 by Tom Preston-Werner, Chris Wanstrath, PJ Hyett.

Some of the other platforms similar to github are Redmine, Bitbucket, GitLab, and Launchpad.

I would use such a platform because, with a community of more than 11 million people, developers can discover, use, and contribute to over 27 million projects using a powerful collaborative development workflow. Using GitHub.com one can integrate GitHub with third party tools, from project management to continuous deployment, to build software in the way that works best for her.

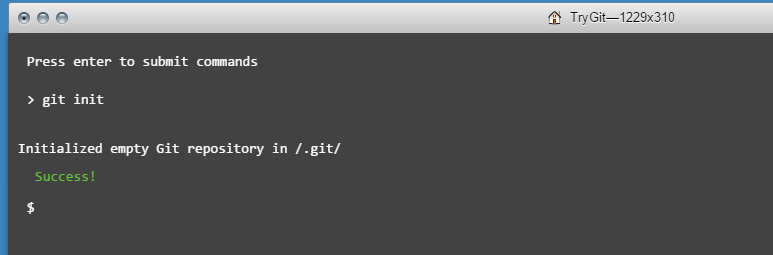
**GIT Tutorial**

1.1 Got 15 minutes and want to learn Git?

Git allows groups of people to work on the same documents (often code) at the same time, and without stepping on each other's toes. It's a distributed version control system.

Our terminal prompt below is currently in a directory we decided to name "octobox". To initialize a Git repository here, type the following command:

git init



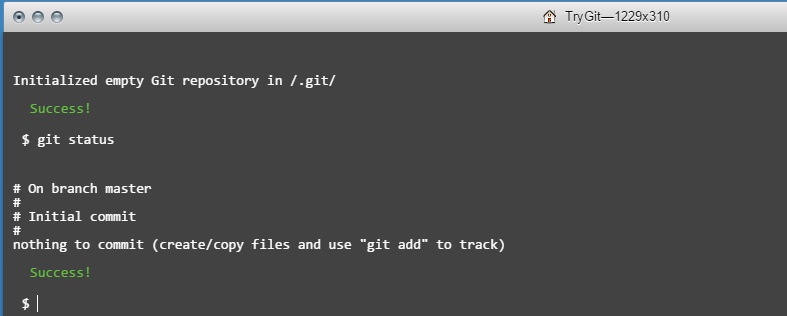
# 1.2 Checking the Status

Good job! As Git just told us, our "octobox" directory now has an empty repository in /.git/. The repository is a hidden directory where Git operates.

To save your progress as you go through this tutorial -- and earn a badge when you successfully complete it -- head over to[create a free Code School account](https://www.codeschool.com/account/courses/try-git/add). We'll wait for you here.

Next up, let's type the git status command to see what the current state of our project is:

git status

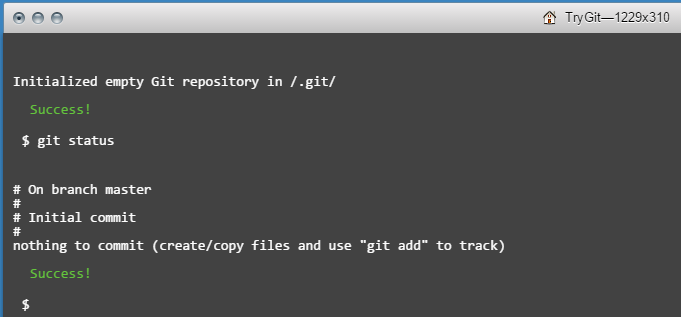


# 1.3 Adding & Committing

I created a file called octocat.txt in the octobox repository for you (as you can see in the browser below).

You should run the git status command again to see how the repository status has changed:

git status

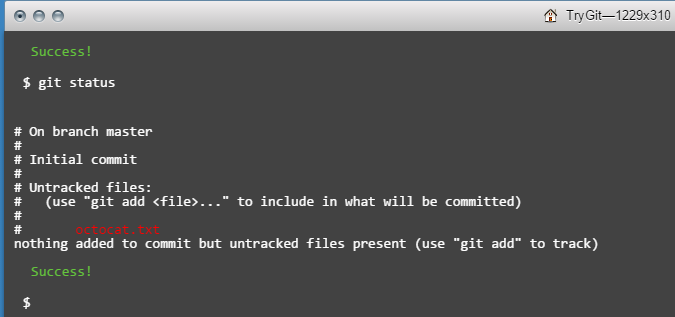


# 1.4 Adding Changes

Good, it looks like our Git repository is working properly. Notice how Git says octocat.txt is "untracked"? That means Git sees that octocat.txt is a new file.

To tell Git to start tracking changes made to octocat.txt, we first need to add it to the staging area by using git add.

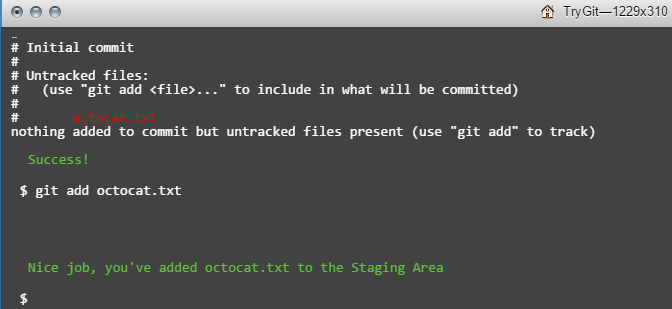
git add octocat.txt



# 1.5 Checking for Changes

Good job! Git is now tracking our octocat.txt file. Let's run git status again to see where we stand:

git status

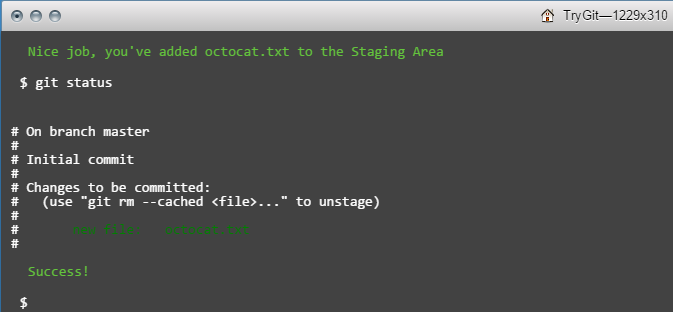


# 1.6 Committing

Notice how Git says changes to be committed? The files listed here are in the Staging Area, and they are not in our repository yet. We could add or remove files from the stage before we store them in the repository.

To store our staged changes we run the commit command with a message describing what we've changed. Let's do that now by typing:

git commit -m "Add cute octocat story"

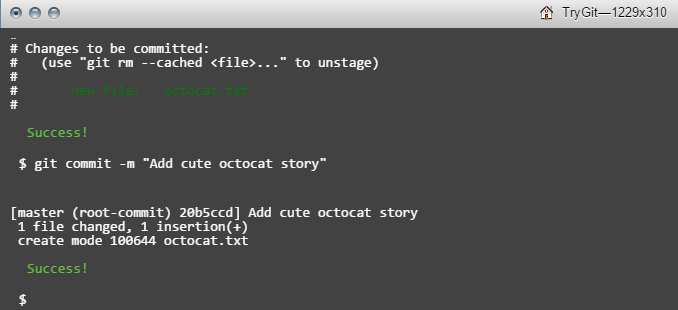


# 1.7 Adding All Changes

Great! You also can use wildcards if you want to add many files of the same type. Notice that I've added a bunch of .txt files into your directory below.

I put some in a directory named "octofamily" and some others ended up in the root of our "octobox" directory. Luckily, we can add all the new files using a wildcard with git add. Don't forget the quotes!

git add '\*.txt'

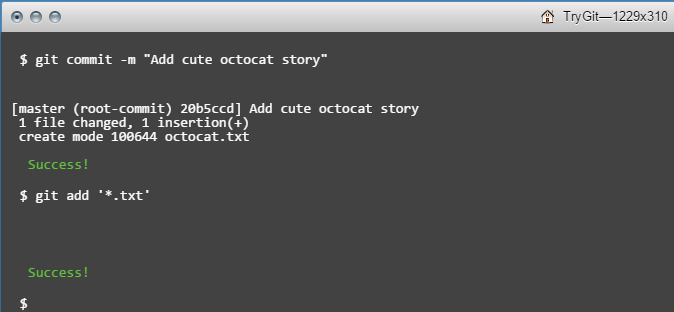


# 1.8 Committing All Changes

Okay, you've added all the text files to the staging area. Feel free to run git status to see what you're about to commit.

If it looks good, go ahead and run:

git commit -m 'Add all the octocat txt files'

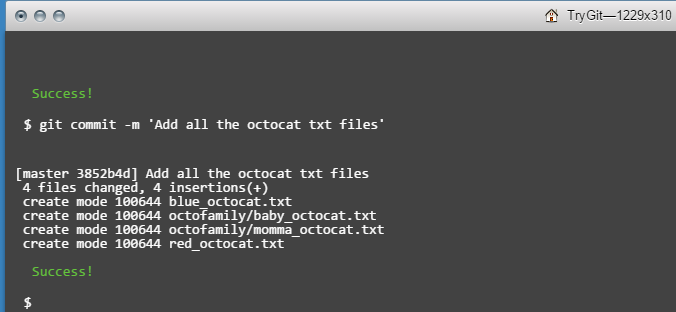


# 1.9 History

So we've made a few commits. Now let's browse them to see what we changed.

Fortunately for us, there's git log. Think of Git's log as a journal that remembers all the changes we've committed so far, in the order we committed them. Try running it now:

git log



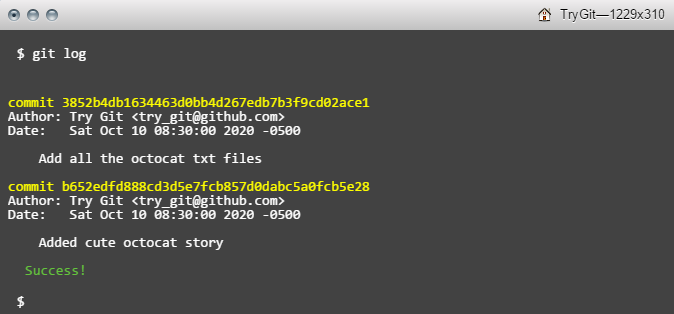
# 1.10 Remote Repositories

Great job! We've gone ahead and created a new empty GitHub repository for you to use with Try Git athttps://github.com/try-git/try\_git.git. To push our local *repo* to the GitHub server we'll need to add a remote repository.

This command takes a *remote name* and a *repository URL*, which in your case is https://github.com/try-git/try\_git.git.

Go ahead and run git remote add with the options below:

git remote add origin https://github.com/try-git/try\_git.git

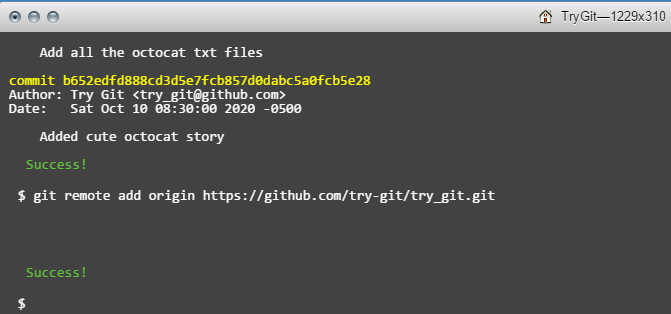


# 1.11 Pushing Remotely

The push command tells Git where to put our commits when we're ready, and boy we're ready. So let's push our local changes to our **origin** repo (on GitHub).

The name of our remote is origin and the default local branch name is master. The -u tells Git to remember the parameters, so that next time we can simply run git push and Git will know what to do. Go ahead and push it!

git push -u origin master

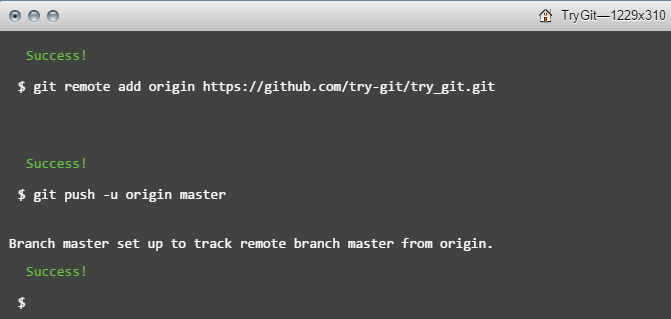


# 1.12 Pulling Remotely

Let's pretend some time has passed. We've invited other people to our GitHub project who have pulled your changes, made their own commits, and pushed them.

We can check for changes on our GitHub repository and pull down any new changes by running:

git pull origin master

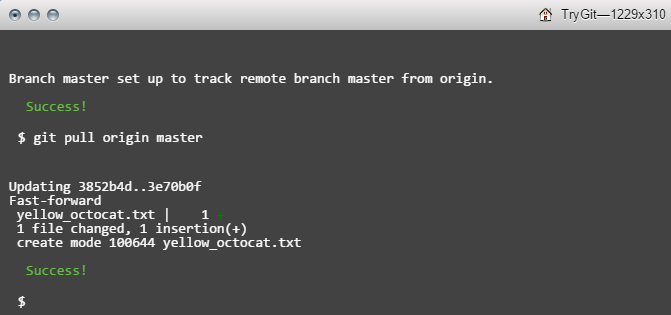


# 1.13 Differences

Uh oh, looks like there have been some additions and changes to the octocat family. Let's take a look at what is differentfrom our last commit by using the git diff command.

In this case we want the diff of our most recent commit, which we can refer to using the HEAD pointer.

git diff HEAD

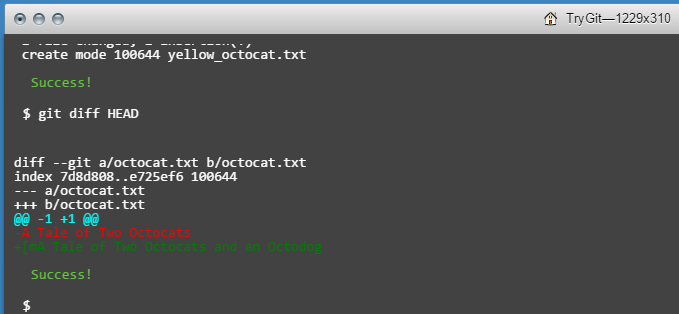


# 1.14 Staged Differences

Another great use for diff is looking at changes within files that have already been staged. Remember, staged files are files we have told git that are ready to be committed.

Let's use git add to stage octofamily/octodog.txt, which I just added to the family for you.

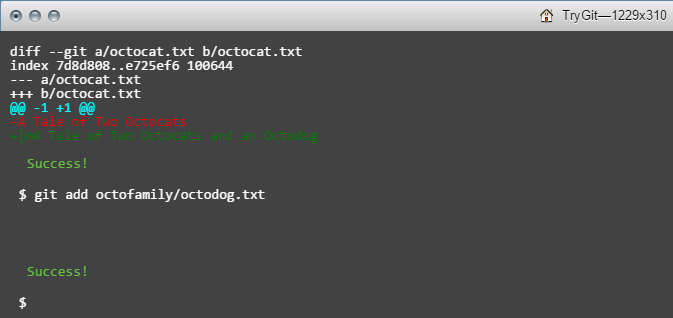
git add octofamily/octodog.txt



# 1.15 Staged Differences (cont'd)

Good, now go ahead and run git diff with the --staged option to see the changes you just staged. You should see thatoctodog.txt was created.

git diff --staged

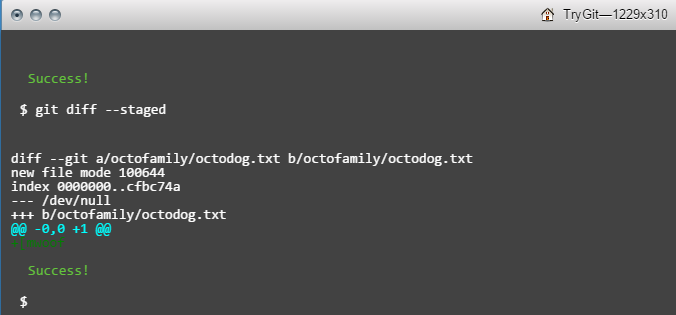


# 1.16 Resetting the Stage

So now that octodog is part of the family, octocat is all depressed. Since we love octocat more than octodog, we'll turn his frown around by removing octodog.txt.

You can unstage files by using the git reset command. Go ahead and remove octofamily/octodog.txt.

git reset octofamily/octodog.txt

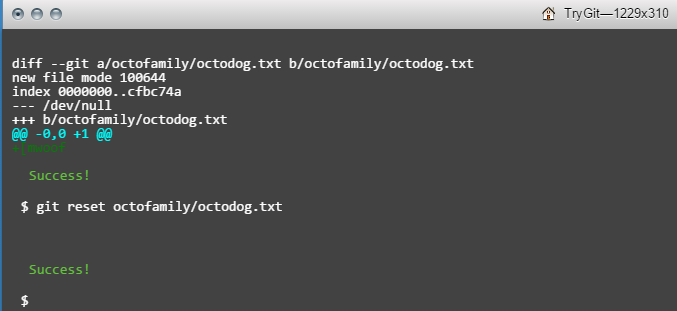


# 1.17 Undo

git reset did a great job of unstaging octodog.txt, but you'll notice that he's still there. He's just not staged anymore. It would be great if we could go back to how things were before octodog came around and ruined the party.

Files can be changed back to how they were at the last commit by using the command: git checkout -- <target>. Go ahead and get rid of all the changes since the last commit for octocat.txt

git checkout -- octocat.txt

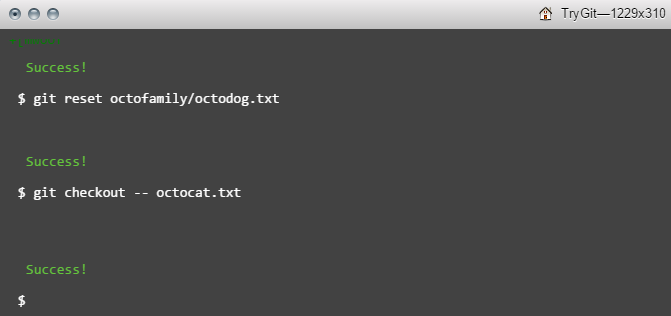


# 1.18 Branching Out

When developers are working on a feature or bug they'll often create a copy (aka. branch) of their code they can make separate commits to. Then when they're done they can merge this branch back into their main master branch.

We want to remove all these pesky octocats, so let's create a branch called clean\_up, where we'll do all the work:

git branch clean\_up

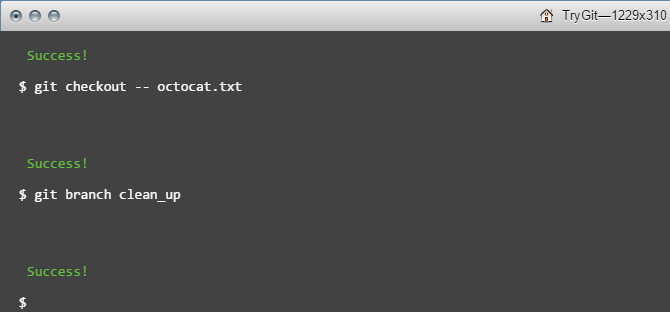


# 1.19 Switching Branches

Great! Now if you type git branch you'll see two local branches: a main branch named master and your new branch namedclean\_up.

You can switch branches using the git checkout <branch> command. Try it now to switch to the clean\_up branch:

git checkout clean\_up

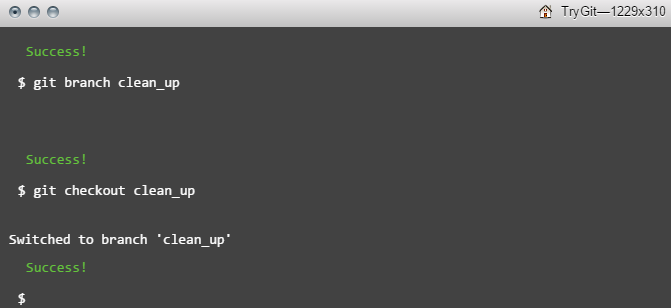


# 1.20 Removing All The Things

Ok, so you're in the clean\_up branch. You can finally remove all those pesky octocats by using the git rm command which will not only remove the actual files from disk, but will also stage the removal of the files for us.

You're going to want to use a wildcard again to get all the octocats in one sweep, go ahead and run:

git rm '\*.txt'

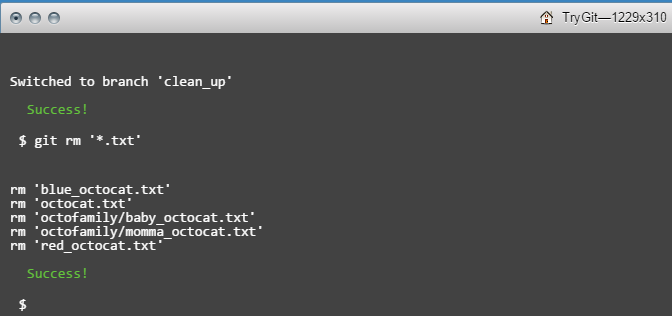


# 1.21 Commiting Branch Changes

Now that you've removed all the cats you'll need to commit your changes.

Feel free to run git status to check the changes you're about to commit.

git commit -m "Remove all the cats"

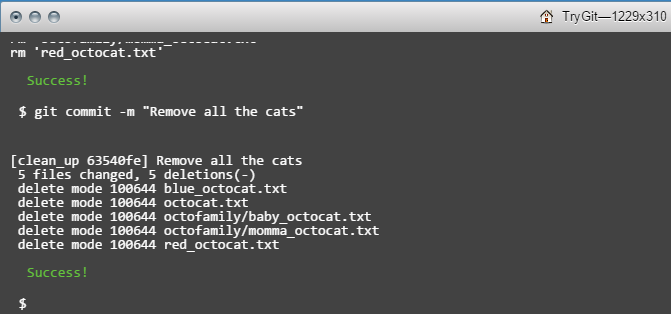


# 1.22 Switching Back to master

Great, you're almost finished with the cat... er the bug fix, you just need to switch back to the master branch so you can copy (or merge) your changes from the clean\_up branch back into the master branch.

Go ahead and checkout the master branch:

git checkout master

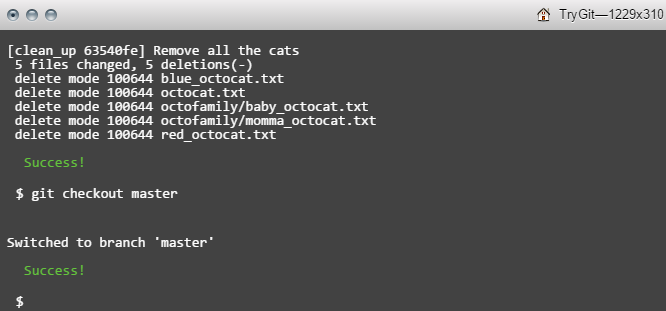


# 1.23 Preparing to Merge

Alrighty, the moment has come when you have to merge your changes from the clean\_up branch into the master branch. Take a deep breath, it's not that scary.

We're already on the master branch, so we just need to tell Git to merge the clean\_up branch into it:

git merge clean\_up

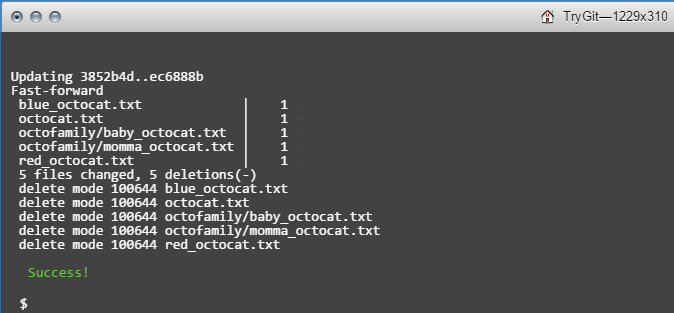


# 1.24 Keeping Things Clean

Congratulations! You just accomplished your first successful bugfix and merge. All that's left to do is clean up after yourself. Since you're done with the clean\_up branch you don't need it anymore.

You can use git branch -d <branch name> to delete a branch. Go ahead and delete the clean\_up branch now:

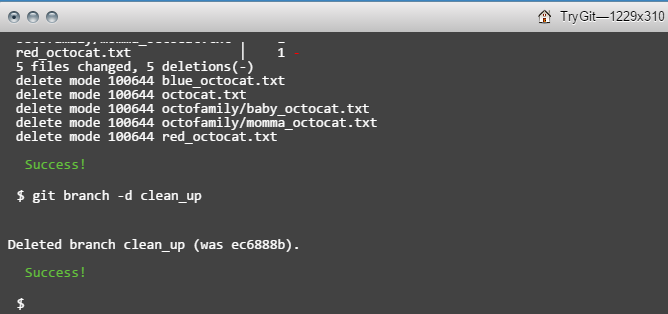
git branch -d clean\_up



# 1.25 The Final Push

Here we are, at the last step. I'm proud that you've made it this far, and it's been great learning Git with you. All that's left for you to do now is to push everything you've been working on to your remote repository, and you're done!

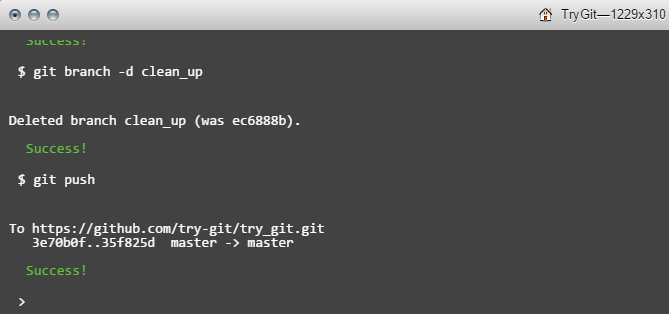
git push



# 1.25 The Final Push

Great! You now have a little taste of the greatness of Git. You can take a look at the wrap up page for a little more information on Git and GitHub, oh, and of course your badge!

[Wrap it all Up](https://try.github.io/wrap_up)



**Define the following terms in the context of Git:**

**Repository:** A repository is the most basic element of GitHub and they're easiest to imagine as a project's folder. A repository contains all of the project files (including documentation), and stores each file's revision history; Repositories can have multiple collaborators and can be either public or private.

**Commit:** A commit, or "revision", is an individual change to a file (or set of files). It's like when you save a file, except with Git, every time you save it creates a unique ID (a.k.a. the "SHA" or "hash") that allows you to keep record of what changes were made when and by who, and Commits usually contain a commit message which is a brief description of what changes were made.

**Push:** Pushing refers to sending your committed changes to a remote repository such as GitHub.com. For instance, if you change something locally, you'd want to then push those changes so that others may access them.

**Branch:** A branch is a parallel version of a repository - it is contained within the repository, but does not affect the primary or master branch allowing you to work freely without disrupting the "live" version. When you've made the changes you want to make, you can merge your branch back into the master branch to publish your changes.

**Fork:** A fork is a personal copy of another user's repository that lives on your account, and Forks allow you to freely make changes to a project without affecting the original. Forks remain attached to the original, allowing you to submit a pull request to the original's author to update with your changes; you can also keep your fork up to date by pulling in updates from the original.

**Merge:** Merging takes the changes from one branch (in the same repository or from a fork), and applies them into another. This often happens as a Pull Request (which can be thought of as a request to merge), or via the command line.

**Clone:** A clone is a copy of a repository that lives on your computer instead of on a website's server somewhere, or the act of making that copy. With your clone you can edit the files in your preferred editor and use Git to keep track of your changes without having to be online, and you can push your local changes to the remote to keep them synced when you're online.

**Pull:** Pull refers to when you are fetching in changes and merging them. For instance, if someone has edited the remote file you're both working on, you'll want to pull in those changes to your local copy so that it's up to date.

**Pull Request:** Pull requests are proposed changes to a repository submitted by a user and accepted or rejected by a repository's collaborators. Like issues, pull requests each have their own discussion forum.