**git --version**

You can initialize your current working directory as a Git repository with init.

**git init**

To copy an existing Git repository hosted remotely, you’ll use git clone with the repo’s URL or server location (in the latter case you will use ssh).

**git clone https://www.github.com/username/repo-name**

Show your current Git directory’s remote repository.

**git remote**

For a more verbose output, use the -v flag.

**git remote -v**

Add the Git upstream, which can be a URL or can be hosted on a server (in the latter case, connect with ssh).

**git remote add upstream https://www.github.com/username/repo-name**

Staging

**When you’ve modified a file and have marked it to go in your next commit, it is considered to be a staged file.**

Check the status of your Git repository, including files added that are not staged, and files that are staged.

**git status**

To stage modified files, use the **add** command, which you can run multiple times before a commit. If you make subsequent changes that you want included in the next commit, you must run add again.

You can specify the specific file with add.

**git add my\_script.py**

With . you can add all files in the current directory including files that begin with a ..

git add .

You can remove a file from staging while retaining changes within your working directory with reset.

**git reset my\_script.py**

Committing

**Once you have staged your updates, you are ready to commit them, which will record changes you have made to the repository.**

To commit staged files, you’ll run the commit command with your meaningful commit message so that you can track commits.

**git commit -m "Commit message"**

You can condense staging all tracked files with committing them in one step.

**git commit -am "Commit message"**

If you need to modify your commit message, you can do so with the --amend flag.

**git commit --amend -m "New commit message"**

**Branches**

A branch in Git is a movable pointer to one of the commits in the repository, it allows you to isolate work and manage feature development and integrations. You can learn more about branches by reading the Git documentation.

List all current branches with the branch command. An asterisk (\*) will appear next to your currently active branch.

**git branch**

Create a new branch. You will remain on your currently active branch until you switch to the new one.

**git branch new-branch**

Switch to any existing branch and check it out into your current working directory.

**git checkout another-branch**

You can consolidate the creation and checkout of a new branch by using the -b flag.

**git checkout -b new-branch**

Rename your branch name.

git branch -m current-branch-name new-branch-name

Merge the specified branch’s history into the one you’re currently working in.

**git merge branch-name**

Abort the merge, in case there are conflicts.

**git merge --abort**

You can also select a particular commit to merge with cherry-pick with the string that references the specific commit.

**git cherry-pick f7649d0**

When you have merged a branch and no longer need the branch, you can delete it.

**git branch -d branch-name**

If you have not merged a branch to master, but are sure you want to delete it, you can force delete a branch.

**git branch -D branch-name**

Collaborate and Update

To download changes from another repository, such as the remote upstream, you’ll use fetch.

**git fetch upstream**

Merge the fetched commits.

**git merge upstream/master**

Push or transmit your local branch commits to the remote repository branch.

**git push origin master**Fetch and merge any commits from the tracking remote branch.

**git pull**

Inspecting

Display the commit history for the currently active branch.

**git log**

Show the commits that changed a particular file. This follows the file regardless of file renaming.

**git log --follow my\_script.py**

Show the commits that are on one branch and not on the other. This will show commits on a-branch that are not on b-branch.

**git log a-branch..b-branch**

Look at reference logs (reflog) to see when the tips of branches and other references were last updated within the repository.

**git reflog**

Show any object in Git via its commit string or hash in a more human-readable format.

**git show de754f5**

Show Changes

The git diff command shows changes between commits, branches, and more. You can read more fully about it through the Git documentation.

Compare modified files that are on the staging area.

**git diff --staged**

Display the diff of what is in a-branch but is not in b-branch.

**git diff a-branch..b-branch**

Show the diff between two specific commits.

**git diff 61ce3e6..e221d9c**

**Stashing**

Sometimes you’ll find that you made changes to some code, but before you finish you have to begin working on something else. You’re not quite ready to commit the changes you have made so far, but you don’t want to lose your work. The git stash command will allow you to save your local modifications and revert back to the working directory that is in line with the most recent HEAD commit.

Stash your current work.

**git stash**

See what you currently have stashed.

**git stash list**

Your stashes will be named stash@{0}, stash@{1}, and so on.

Show information about a particular stash.

**git stash show stash@{0}**

To bring the files in a current stash out of the stash while still retaining the stash, use apply.

**git stash apply stash@{0}**

If you want to bring files out of a stash, and no longer need the stash, use pop.

**git stash pop stash@{0}**

If you no longer need the files saved in a particular stash, you can drop the stash.

**git stash drop stash@{0}**

If you have multiple stashes saved and no longer need to use any of them, you can use clear to remove them.

**git stash clear**

Ignoring Files

If you want to keep files in your local Git directory, but do not want to commit them to the project, you can add these files to your .gitignore file so that they do not cause conflicts.

Use a text editor such as nano to add files to the .gitignore file.

nano .gitignore

To see examples of .gitignore files, you can look at GitHub’s .gitignore template repo.

**Rebasing**

A rebase allows us to move branches around by changing the commit that they are based on. With rebasing, you can squash or reword commits.

You can start a rebase by either calling the number of commits you have made that you want to rebase (5 in the case below).

**git rebase -i HEAD~5**

Alternatively, you can rebase based on a particular commit string or hash.

**git rebase -i 074a4e5**

Once you have squashed or reworded commits, you can complete the rebase of your branch on top of the latest version of the project’s upstream code.

**git rebase upstream/master**

To learn more about rebasing and updating, you can read How To Rebase and Update a Pull Request, which is also applicable to any type of commit.

**Resetting**

Sometimes, including after a rebase, you need to reset your working tree. You can reset to a particular commit, and delete all changes, with the following command.

**git reset --hard 1fc6665**

To force push your last known non-conflicting commit to the origin repository, you’ll need to use --force.

Warning: Force pushing to master is often frowned upon unless there is a really important reason for doing it. Use sparingly when working on your own repositories, and work to avoid this when you’re collaborating.

**git push --force origin master**

To remove local untracked files and subdirectories from the Git directory for a clean working branch, you can use git clean.

**git clean -f -d**

If you need to modify your local repository so that it looks like the current upstream master (that is, there are too many conflicts), you can perform a hard reset.

Note: Performing this command will make your local repository look exactly like the upstream. Any commits you have made but that were not pulled into the upstream will be destroyed.

**git reset --hard upstream/master**

Conclusion

This guide covers some of the more common Git commands you may use when managing repositories and collaborating on software.

You can learn more about open-source software and collaboration in our Introduction to Open Source tutorial series:

git --help

Create a New Branch

Whenever you work on a collaborative project, you and other programmers contributing to the repository will have different ideas for new features or fixes at once. Some of these new features will not take significant time to implement, but some of them will be ongoing. Because of this, it is important to branch the repository so that you are able to manage the workflow, isolate your code, and control what features make it back to the main branch of the project repository.

The default main branch of a project repository is usually called the master branch. A common best practice is to consider anything on the master branch as being deployable for others to use at any time.

When creating a branch, it is very important that you create your new branch off of the master branch. You should also make sure that your branch name is a descriptive one. Rather than calling it my-branch, you should go with frontend-hook-migration or fix-documentation-typos instead.

cd repository

Now, we’ll create our new branch with the **git branch** command. Make sure you name it descriptively so that others working on the project understand what you are working on.

**git branch new-branch**

Now that our new branch is created, we can switch to make sure that we are working on that branch by using the git checkout command:

**git checkout new-branch**

Once you enter the git checkout command, you will receive the following output:

Output

Switched to branch 'new-branch'

Alternatively, you can condense the above two commands, creating and switching to a new branch, with the following command and -b flag:

**git checkout -b new-branch**

If you want to switch back to master, you’ll use the checkout command with the name of the master branch:

**git checkout master**

The checkout command will allow you to switch between multiple branches, so you can potentially work on multiple features at once.

At this point, you can now modify existing files or add new files to the project on your own branch.

**Make Changes Locally**

Once you have modified existing files or added new files to the project, you can add them to your local repository, which we can do with the git add command. Let’s add the -A flag to add all changes that we have made:

**git add -A**

Next, we’ll want to record the changes that we made to the repository with the git commit command.

The **commit message** is an important aspect of your code contribution; it helps the other contributors fully understand the change you have made, why you made it, and how significant it is. Additionally, commit messages provide a historical record of the changes for the project at large, helping future contributors along the way.

If we have a very short message, we can record that with the -m flag and the message in quotes:

**git commit -m "Fixed documentation typos"**

But, unless it is a very minor change, we will more than likely want to include a lengthier commit message so that our collaborators are fully up to speed with our contribution. To record this larger message, we will run the git commit command which will open the default text editor:

**git commit**

If you would like to configure your default text editor, you can do so with the git config command, and set nano as the default editor, for example:

**git config --global core.editor "nano"**

Or vim:

**git config --global core.editor "vim"**

After running the git commit command, depending on the default text editor you’re using, your terminal window should display a document ready for you to edit that will look similar to this:

GNU nano 2.0.6 File: …username/repository/.git/COMMIT\_EDITMSG

# Please enter the commit message for your changes. Lines starting

# with '#' will be ignored, and an empty message aborts the commit.

# On branch new-branch

# Your branch is up-to-date with 'origin/new-branch'.

#

# Changes to be committed:

# modified: new-feature.py

#

Underneath the introductory comments, you should add the commit message to the text file.

To write a useful commit message, you should include a summary on the first line that is around 50 characters long. Under this, and broken up into digestible sections, you should include a description that states the reason you made this change, how the code works, and additional information that will contextualize and clarify it for others to review the work when merging it. Try to be as helpful and proactive as possible to ensure that those maintaining the project are able to fully understand your contribution.

Once you have saved and exited the commit message text file, you can verify what git will be committing with the following command:

**git status**

Depending on the changes that you have made, you will receive output that looks something like this:

Output

On branch new-branch

Your branch is ahead of 'origin/new-branch' by 1 commit.

(use "git push" to publish your local commits)

nothing to commit, working directory clean

At this point you can use the git push command to push the changes to the current branch of your forked repository:

**git push --set-upstream origin new-branch**

The command will provide you with output to let you know of the progress, and it will look similar to the following:

Output

Counting objects: 3, done.

Delta compression using up to 4 threads.

Compressing objects: 100% (2/2), done.

Writing objects: 100% (3/3), 336 bytes | 0 bytes/s, done.

Total 3 (delta 0), reused 0 (delta 0)

To https://github.com/your-username /respository .git

a1f29a6..79c0e80 new-branch -> new-branch

Branch new-branch set up to track remote branch new-branch from origin.

You can now navigate to the forked repository on your GitHub webpage and toggle to the branch you just pushed to see the changes you have made in-browser.

At this point, it is possible to make a pull request to the original repository, but if you have not already done so, you’ll want to make sure that your local repository is up-to-date with the upstream repository.

Update Local Repository

While you are working on a project alongside other contributors, it is important for you to keep your local repository up-to-date with the project as you don’t want to make a pull request for code that will cause conflicts. To keep your local copy of the code base updated, you’ll need to sync changes.

We’ll first go over configuring a remote for the fork, then syncing the fork.

Configure a Remote for the Fork

Remote repositories make it possible for you to collaborate with others on a Git project. Each remote repository is a version of the project that is hosted on the Internet or a network you have access to. Each remote repository should be accessible to you as either read-only or read-write, depending on your user privileges.

In order to be able to sync changes you make in a fork with the original repository you’re working with, you need to configure a remote that references the upstream repository. You should set up the remote to the upstream repository only once.

Let’s first check which remote servers you have configured. The git remote command will list whatever remote repository you have already specified, so if you cloned your repository as we did above, you’ll at least see the origin repository, which is the default name given by Git for the cloned directory.

From the directory of the repository in our terminal window, let’s use the git remote command along with the -v flag to display the URLs that Git has stored along with the relevant remote shortnames (as in “origin”):

**git remote -v**

Since we cloned a repository, our output should look similar to this:

Output

origin https://github.com/your-username/forked-repository.git (fetch)

origin https://github.com/your-username/forked-repository.git (push)

If you have previously set up more than one remote, the git remote -v command will provide a list of all of them.

Next, we’ll specify a new remote upstream repository for us to sync with the fork. This will be the original repository that we forked from. We’ll do this with the git remote add command.

**git remote add upstream https://github.com/original-owner-username/original-repository.git**

In this example, upstream is the shortname we have supplied for the remote repository since in terms of Git, “upstream” refers to the repository that we cloned from. If we want to add a remote pointer to the repository of a collaborator, we may want to provide that collaborator’s username or a shortened nickname for the shortname.

We can verify that our remote pointer to the upstream repository was properly added by using the git remote -v command again from the repository directory:

**git remote -v**

Output

origin https://github.com/your-username/forked-repository.git (fetch)

origin https://github.com/your-username/forked-repository.git (push)

upstream https://github.com/original-owner-username/original-repository.git (fetch)

upstream https://github.com/original-owner-username/original-repository.git (push)

Now you can refer to upstream on the command line instead of writing the entire URL, and you are ready to sync your fork with the original repository.

Sync the Fork

Once we have configured a remote that references the upstream and original repository on GitHub, we are ready to sync our fork of the repository to keep it up-to-date.

To sync our fork, from the directory of our local repository in a terminal window, we’ll use the git fetch command to fetch the branches along with their respective commits from the upstream repository. Since we used the shortname “upstream” to refer to the upstream repository, we’ll pass that to the command:

**git fetch upstream**

Depending on how many changes have been made since we forked the repository, your output may be different, and may include a few lines on counting, compressing, and unpacking objects. Your output will end similarly to the following lines, but may vary depending on how many branches are part of the project:

\* [new branch] master -> upstream/master

Now, commits to the master branch will be stored in a local branch called upstream/master.

Let’s switch to the local master branch of our repository:

**git checkout master**

Output

Switched to branch 'master'

We’ll now merge any changes that were made in the original repository’s master branch, that we will access through our local upstream/master branch, with our local master branch:

**git merge upstream/master**

The output here will vary, but it will begin with Updating if changes have been made, or Already up-to-date. if no changes have been made since you forked the repository.

Your fork’s master branch is now in sync with the upstream repository, and any local changes you made were not lost.

Depending on your own workflow and the amount of time you spend on making changes, you can sync your fork with the upstream code of the original repository as many times as it makes sense for you. But you should certainly sync your fork right before making a pull request to make sure you don’t contribute conflicting code.

Create Pull Request

At this point, you are ready to make a pull request to the original repository.

You should navigate to your forked repository, and press the “New pull request” button on your left-hand side of the page.