Git tutorial

How to use git.

Table of contents

1. ~~Getting~~ Gitting Started

Initialize a repository and make your first commit.

About this tutorial

This tutorial covers git for command line, and assumes that you have git installed on your computer. If you do not, you can find it here.

On Windows, make sure to include the command-line tools and enable support for MinTTY.

It will also assume you have some folder that you will use as your "project folder" for the purpose of this tutorial. It's okay if it is empty.

Opening the project folder

Git repositories are stored within your project's main folder or **root directory**. To start a repository, you will need to open your project's root directory from a terminal.

OSX & Linux

On Mac, you can find the application called "Terminal" in your Applications folder. On Linux, it varies, but you can probably find in the HUD by pressing the Super key and searching for "Terminal," or directly by pressing `Ctrl-Alt-T` (this typically works on Ubuntu-based systems, at least).

When you have the terminal open, **change directories** into the project's root. To do this, type `cd path/to/project/directory`, replacing `path/to/project/directory` with the actual path to the directory. Hint: Your terminal most likely opens to a directory called `~`, also known as the **home directory**, which contains folders like `Documents` and `Desktop`. You can type `ls` to see what is in a folder at any time. You can also type the first few characters (to the point where the name is unique) and hit the tab key to complete it. You may also be able to drag-and-drop the name of the folder from the file explorer (e.g., Finder or Nautilus).

Windows

Open the project folder in your file explorer. Right click, and select "Git Bash here."

Initializing a repository

Now that you are in the project's root directory, you can initialize a git repository. To do this enter into the command line:

```
git init
```

Tada! You have now initialized a blank git repository.

Stage some files

A git repository tracks and records changes to files. When you have a file in your directory that you want to keep track of, you will need to **add** it to the **staging area** and **commit** it to the repository.

First, let's make a file to stage. In a text editor, create a file in your project's root called `something-important.txt` (realistically, you can call it whatever you want). Write something in it, like "This is some important information that I want to keep track of." Then save it.

Now, in your terminal, enter:

```
git status
```

You should see `something-important.txt` under "Untracked files." In order for git to keep track of it, you'll need to **stage** it. Enter:

```
git add something-important.txt
```

Now check the status again:

```
git status
```

You should see your file under "Changes to be committed." That means that git has made a note of the current state of your file. If you make any more changes to it that you want to commit, you'll need to `add` it again. You can stage as many files as you want prior to a commit.

Making your initial commit

A **commit** is sort of like an entry in a log. You have gathered up all of your relevant changes, staged them, and now you are ready to create a snapshot of their current state before moving on. When you make a commit, you will write a brief **commit message** about what you changed. Then, at any point in the future, you can come back to that commit, see what was different between before and after the commit, and see what you said about it. This is really helpful if you can't remember why you added something, can't decipher an old comment, can't figure out what happened to a few lines—basically, if you're ever coding while tired.

So let's make that first commit. By convention, the message on your first commit to a repository is usually "Initial commit." To make a commit, you run the command `git commit` with the **option** `-m` followed by your commit message in quotes.

```
git commit -m "Initial commit."
```

Congratulations! You've just made your first commit to a repository. See 02_workflow.md for the next steps.

2. Workflow

Now that you've made your first commit, you'll want to get into the habit of staging and committing changes as you go.

Staging more changes

Whether the file is brand new, or already being tracked but just modified, you can add it to the staging area with `git add [filename].[ext]` (without the brackets). You can also list as many files as you want here, separated by spaces, and you can use regular expressions. So something like this would be valid:

```
git add file1.txt file2.txt data/*.csv
```

Remember that you can always see what is staged by running `git status`.

You don't have to add everything to the staging area if you don't want to commit its changes. So, if you have files that you don't want tracked (see "Ignoring files" below), or if you've made changes that you think don't really fit with the theme of your current commit, you can just not stage them, and then they won't be included in your commit.

On the flip side, if you forget to stage a file, it won't be included in your commit, so double check the `status` before you make a commit just to be safe.

More commits

You should make a commit every time you are *about to* change something big, and again afterward. That way, if you mess something up, you can just roll back in time and pretend it never happened.

Skipping the staging (or, staging at the commit)

Sometimes, even most of the time, you might not need to do too much with staging. You might be only working on existing files rather than adding new ones, and committing all changes to tracked files at every commit. In these situations, it is possible to skip the `git add` parts. Instead, run your `commit` with the option `-a` for "all." You can combine single-letter options, as long as any with a parameter are last. That means, this:

```
git commit -a -m "My commit message."
```

is the same as

```
git commit -am "My commit message."
```

Commit messages

Commit messages are generally short (~120 characters max), imperative ("Update README.md." instead of "Update**d** README.md."), and end with a period. It is important to be as descriptive as possible in your commit message. This is especially important when collaborating or working on an open source project. If you cannot fit all of the necessary information in your commit message, you can use the extended version.

To write an extended commit message, make your commit without the `-m "Message"` option. That is, just run `git commit` or `git commit -a`. This will open a text editor for you to enter your message.

On the first line, write your brief summary, but end it with "…" For example, "Make important changes to my_script.py…"

Then skip a line.

Then, begin your extended message. Here, you can be as detailed as you need to be about the changes that you made. You don't have to use the imperative. Try to cover the rationale for the changes, as well as the outcome (Was it successful? Is it a work in progress?).

When you're done, save it and exit. This will take you back to the command line and finish the commit.

Ignoring files

Sometimes you will have files that you do not want to track. They might be logs, output from programs, or contain sensitive information. It is a little annoying to have these files show up when you run `git status`, and it can clutter up the visual space and make you miss important files that you *did* mean to commit. Fortunately, `.gitignore` exists.

First, let's make a file to ignore. In a text editor, create a file called something like `ignore-me.txt`, and have it say something like "This is some sensitive data that should not end up in a repository!" (If you're feeling fancy, run `echo "This is some sensitive data that should not end up in a  repository!" > ignore-me.txt` instead.)

Now run `git status`. You'll see that file listed. Let's change that.

In a text editor, create a file called `.gitignore` — the "." is important!

Write `ignore-me.txt` in it. You can add as many files as you want, as well as regular expressions (e.g., `*.log`), separated by new lines. When you're done, save and exit.

Now run `git status` again. You shouldn't see `ignore-me.txt` listed anymore, but you should see `.gitignore`. It is customary to commit `.gitignore` to your repository so that in the future, you or others won't have to re-write it.

`.gitignore` works just like any other file in terms of git workflow. When you change it, you will want to stage and commit it.

3. Working with remotes

One great thing about git is easy to host your repositories on a **remote server** (online). This makes it easy for you to access it from anywhere, as well as for others to collaborate while keeping track of who made what changes.

Github and Bitbucket are two major sites that host git repositories. This tutorial will mostly talk about Github, but the process is more or less the same for any remote.

Adding a remote

In order to host your repository on a remote server, you first need to set up a repository on that server. On Github, you would click the "+" button in the upper right, and select "New repository." Name it whatever you want, select "public" or "private" as you see fit, and ignore the options to initialize with a README or license; those don't apply when you already have a repository that you are going to push out.

Once it is created, it will bring you to a page that lists several URLs. Copy the HTTPS one. For the sake of example, let's say it's `https://github.com/example/example-repo.git`.

Now go over to your terminal and enter:

```
git remote add origin https://github.com/example/example-repo.git
```

Let's break that down. `git` is what tells it that you're doing a git command. `remote` says that you are going to be telling it to do something with a remote server. `add` says that specifically, you are *adding* a remote. `origin` is the name of the new remote that you are adding. You get to pick the name—it could really be anything. "origin" is just the convention. And then the URL tells it where to point when fetching or pushing changes on origin.

So now your git repository knows of a remote server, located at `https://github.com/example/example-repo.g`
and knows that that's what you're referring to when you talk about "origin."

Pushing to a remote

Now, you'll want to get your repository onto the remote. This is called `push`ing.

The first time you push, you will want to use

```
git push -u origin master
```

`master` is the name of the **branch** that you are on. Branching will be covered later, but in short, it allows you to copy the repo at a certain point so that you can make changes in one branch but not another. By default, the main branch in a repository is called *master*.

The `-u` option tells it to set up the current branch to track any changes on its remote counterpart, and vice versa. It essentially pairs your local branch `master` with the remote branch `origin/master`. That way, for future pushes, you can just run `git push` by itself without any extra arguments. But nothing bad will happen if you still run `git push [-u] origin master` each time.

Integrating this into your workflow

Keep in mind that `commit`ting and `push`ing are two separate steps. You cannot `push` changes that are not `commit`ted, and simply `commit`ting does not `push`. In that way, you can also save up a bunch of commits and push them in one batch, but typically you would probably just push after each commit.

Review so far

Let's take a break to review the steps so far. Note that the order is mostly only locally-dependent (you have to stage before or during the `commit`, not after), but not so much globally-dependent (you can add a remote server whenever you want, as long as it is before you `push`).

Initialize a repository

- `git init`

Add a remote

- Initialize a repo on the remote (e.g., Github or Bitbucket)
- `git remote add origin https://github.com/example/example-repo.git`

Make your first commits and pushes

- Create some files
- `git add [relevant files]`
- `git commit -m "Initial commit."`
- `git push -u origin master`

Let the workflow flow! (or, `while project != complete`)

- Create and/or edit some files
- `git add [relevant files]`
- `git commit -m "My commit message."`
- `git push`

or

- Edit already-tracked files
- `git commit -am "My commit message."`
- `git push`

Pulling from a remote

When you are collaborating with others on one repository, or working across multiple computers, there will likely be times when the remote contains work that is not in your local repository. If your current branch is already tracking a remote branch (such as if you have already run a `git push -u origin master` once), then all you need to do to retrieve these changes is run:

```
git pull
```

If you have not set up your branch to track one on the remote, it's just a couple more terms:

```
git pull origin master
```

But after that first run, you can do `git pull` from then on.

Cloning a repository

Sometimes, you might be retrieving a repository that you do not have on your local machine. You could run through all the steps of making a blank repo on your computer, adding the remote, and pulling from the remote. But this is all accomplished in one command: `git clone [URL]`.

So, to clone a repository, first go to your terminal and `cd` into a folder where you want the new repository to be—it will clone into its own subfolder, so if you have folder called "Projects" or something, that would be suitable.

Then find the URL of the remote repository. On Github, you can find that by clicking the green "Clone or download" button on a repository's main page, and copying the HTTPS URL.

Then, in your terminal, run `git clone https://github.com/example/example-repo.git`

Now `cd` into the project's folder. (While it is cloning, it will say something like "cloning into example-repo/", so then you can do `cd example-repo`.) This will already have the remote set up as "origin", so no need to add it again. Now you can just pick up the workflow like normal.

That's all for remotes

Next, try 04_branching-forking-merging.md

4. Branching, forking, and merging

Branching

A somewhat more advanced feature of git, but still very important, is branching. Branching allows you to duplicate a repository at one point in history, and make changes in one copy that are not reflected in another. This adds a layer of security and makes it easier to mess around without worrying about breaking your project.

First, you may want to see all of the branches that are currently in your repo. To do this, run

```
git branch
```

You will probably see something like

```
*master
```

That means that "master" is the only branch, and the "*" means it is the one you are currently on.

But what if you wanted to make a new branch? Let's call it "dev" for "development". One way you could do this is to run

```
git branch dev
```

Now, check your list of branches again

```
git branch
```

you'll see something like

```
dev
*master
```

Okay, so you made the branch dev, but you're still on master. That means any changes you make will still be made on master, not dev. To switch over to dev, run

```
git checkout dev
```

now you should see something like

```
Switched to branch `dev`
```

Whereas most git commands make sense in come way, I honestly don't know why `checkout` is so named. But anyway...

There is actually a slightly faster way. People used `git branch [new-branch] && git checkout [new-branch]` enough that it got shortened into one command:

```
git checkout -B dev
```

meaning, "Switch over to the new branch 'dev'." (Replace "dev" with whatever you want the new branch to be called.)

Now, any changes that you make will be kept to the branch dev, and master will be unchanged.

When you want to switch back over to master, just use `git checkout master`.

You can have as many branches as you want, and they can branch off of whatever other branch you want them to.

Remember that when you are pushing out a new branch, just using `git push` won't work. You'll need to use `git push -u origin [branch-name]`.

Forking

Forking is kind of like if `git branch` and `git clone` had a baby. It's sort of branching an entire project.

Normally, forking is something you do on a remote site, like Github (which will again be used for the examples). If someone else has a project that you want to either contribute to, or just have a copy of to make some changes for yourself, you would fork it using the "Fork" button at the upper right of a repository's page (below the upper toolbar). This will make a copy of the repository on your account.

Then, go to the fork on your account and clone it to your computer like normal (find the URL with the "Clone or download" button, then run `git clone URL`).

Now, one of the first things you'll want to do is make a new branch for your changes. This makes it easy to keep track of which changes are yours and which changes are from the maintainer.

You'll want to give the branch a descriptive name. For example, let's say you forked the repo to suggest a new color scheme for the user interface. Run

```
git checkout -B color-scheme
```

The next thing to think about is the remote. By default, your fork on Github is "origin." This makes it easy to push your changes to your fork. But you will probably also want to pull changes in from the "original" one, commonly called "upstream." So go grab the URL from the upstream repo, and run

```
git remote add upstream URL
```

Now, when you want to pull from upstream, run

```
git pull upstream/master color-scheme
```

(to be perfectly honest, I am not 100% sure about this one... I will update it if it turns out to be wrong.)

Merging

Hopefully, changes you make to one branch will eventually be stable enough for you to incorporate them into the main project. To do this, you'll `merge` the development branch with master.

First, make sure you don't have any conflicts, where files on both branches have changed, and git doesn't know which one to keep.

Once you've made sure of that, just run `git merge [from] [to]`, only replace `[from]` with the name of the branch containing the changes, and replace `[to]` with the name of the branch receiving the changes. So that would be something like:

```
git merge dev master
```

Pull requests

When you've made changes to a fork that you would like the maintainer of the upstream repository to consider incorporating into their project, you can submit a **pull request** on that repository's page. If the maintainer likes your changes, they will merge them with (or, "pull them into") their repository. This is primarily how open source development works.

That's all!

Congratulations! That is just about all you need to know to use git.

One last note: There is no need to memorize all of these commands. You might end up memorizing them from using them a lot, but in the meantime, that's what things like `git [command] --help` are for. You can also see glossary.md for a list of the commands covered in this tutorial.

5. Glossary of git commands

`git init`

Initializes a blank repository.
Do this from a project's main folder or **root directory**.

`git status`

Check out what changes have or have not been staged for committing.

`git add [files]`

Stage files to commit.

`git commit -m "My commit message."`

Commit your staged changes.

`git commit -am "My commit message."`

Commit all tracked changes, regardless of whether they have been staged.

`git remote add [name] [URL]`

Add a remote repository located at [URL], called [name].

`git push -u origin master`

Push the branch “master” to the remote “origin”, and set “origin/master” to track the local “master”.

`git push`

Push to the default location (probably `origin/branch-name`).

`git pull`

Pull changes from the remote tracked branch to the current branch.

`git clone [URL]`

Copy the repository located at [URL] here.

`git branch`

List the branches on the current repository.

`git branch branch-name`

Create a new branch called `branch-name`.

`git checkout branch-name`

Switch to the branch `branch-name`.

`git checkout -B branch-name`

Create a branch called `branch-name` and switch to it.

`git merge [from] [to]`

Update branch `to` with the changes found on branch `from`.