An Introduction to Collaborating with Version Control

Created by Brennen Bearnes

https://learn.adafruit.com/an-introduction-to-collaborating-with-version-control

Last updated on 2023-08-29 02:57:49 PM EDT
## Table of Contents

**Overview**
- Version Control? What?
- What's So Special About Git?

**Installing Git**

**Installing Git on Linux**

**Installing Git on Windows**
- Installing msysGit
- Resources

**Installing TortoiseGit on Windows**

**Installing Git on OS X**

**Initializing a Repository and Making Commits**
- Initializing an Empty Repository
- Making a First Commit

**Branches?**

**Publishing Your Repository to GitHub**
- Get You a GitHub Account
- Create a Public Repository and Push Changes

**HTTPS Credential Caching and SSH Keys**
- Cache Username and Password
- Generate an SSH Key

**Working With Existing Repositories**
- Cloning a Repository
- Navigating History
- Assigning Responsibility / History Within a File

**Submitting a Pull Request on GitHub**
- Get a Copy to Work On
- Make Some Changes
- Submit the Pull Request
Overview

We get questions about Git, using Adafruit code published on GitHub, and making pull requests. Version control is a big subject, and this guide is far from comprehensive - it's meant to be a starting place and a pointer to other, deeper resources. We'll update it as things come up.

With that out of the way, let's get started with some basics.

Version Control? What?

If you've ever worked on a long-term project on a computer, especially one that you have to share with other people, there's a pretty good chance you've found yourself looking at something like this:
This has been a problem forever: You've got a project made out of files - something like the source code for an Arduino sketch or a Raspberry Pi project in Python, for example - and as you work on it, you need to know things like:

- What the last stable, known-good version looked like.
- What version is deployed to hardware in the field.
- When, roughly, it was last changed.

Version control systems, also variously known as revision control, source control, and so on, have been around since not long after programmers first started to grapple with this problem. A lot of their core ideas can be found in classic Unix utilities like `diff` and `patch`, designed to examine the differences between individual files or apply changes to them from elsewhere. Nowadays, a respectable VCS can usually:

- Store a project's entire history in a single directory.
- Track exactly how individual files changed, and when.
- Track who they were changed by.
- Log human-readable descriptions of changes.
- Merge sets of changes from different people.
- Display the differences between any two points in a project's history.
- Store multiple concurrent branches of a project, with different changes on each.
Git is the most widely used VCS in the open source community, and GitHub, a web-based platform for hosting Git projects, has become the go-to place for collaborating on source code, supplanting sites like SourceForge and Google Code.

Even though it's not a perfect fit for hardware and design-oriented projects, companies like Adafruit and SparkFun who focus on open source hardware routinely use Git and GitHub to track product changes, publish code and design files, and manage input from the community.

What's So Special About Git?

This section is an alphabet soup of acronyms and things with confusingly similar names. Don't worry about it too much.

Professionals have used tools like SCCS, RCS, and CVS for decades, along with various and sundry commercial offerings. For a lot of users, though, especially small shops and hobbyists, version control didn't really begin to seem easy or straightforward until Subversion, frequently abbreviated "svn", showed up on the scene. Subversion was free, relatively easy to configure, and much less painful to use than earlier systems. It retains one strongly limiting flaw, however: It requires a central server to store changes and share them with others. It operates sort of like a hub and spokes:

Even as svn gained traction with users and gave many their first taste of robust version control, a new wave of open source VCSes came on the scene with a very different approach: Instead of relying on the hub-and-spokes client-server model,
these tools would store a project's complete history locally, in a single repository. For example, with Git, all you have to do to create a new project is this:

```bash
mkdir project
cd project
git init
```

And since every copy of the project is a first-class citizen, with all of the project's history, it's not even strictly necessary to have one central copy of the project (though it's usually convenient). Changes can be moved around from copy to copy, and a "server" can be as simple as having a copy on a computer everyone can access with SSH.

Git, originally written by Linus Torvalds for use in developing the Linux kernel, has more or less taken over the world since its introduction in 2005. That's not to say there aren't other interesting options - tools like Mercurial (), Darcs (), and Bazaar () offer similar features and friendlier interfaces, and are still actively used and maintained - but Git had some advantages in speed and tooling, plus some prominent early users. Once GitHub really started to take off, Git began to seem like the obvious choice to a lot of developers.

As a working programmer, I currently have somewhere around a hundred source code repositories on my laptop. Most of these are tracked with Git:
Among these are applications I routinely use, websites I maintain, variations on the Linux kernel, projects for Adafruit guides, and even the text of books I've written. If it's made out of text files, it's probably useful to keep it in Git. (If it has big binaries in it, this becomes a trickier question - we'll get to that.)

At the time of this writing, Git and GitHub are so widely used that even if you plan to use another system for your own work, knowing the basics is almost certain to be helpful in working with other projects.

Installing Git

Before you begin, there are some things to be conscious of.

Git works most smoothly in a Linux / Unix-like environment. It was first written for use on Linux systems, and still has some expectations that fit best there. This means that if you're running a desktop Linux or OS X, Git and related tools will be easiest to install and configure.

Yes, you can use Git on Windows. Lots of people do, including Lady Ada and (sometimes) yours truly. Just expect that there will be a few more hoops to jump through.

Git is fundamentally a command-line tool. That doesn't mean you have to be a wizard to use it, but it does mean you'll have an easier time if you become comfortable on the command line. Feeling uncertain about this? We've got an entire series on learning Linux with the Raspberry Pi, including guides that focus on command line basics:

- What is the Command Line? ()
- An Illustrated Shell Command Primer ()
There are graphical tools that work with Git repositories, but you will have the easiest time if you install and learn the command-line interface first. Don't worry - it's not actually that bad.

Read on for system-specific instructions.

## Installing Git on Linux

If you're using a mainstream desktop Linux - Debian, Ubuntu, Raspbian, Fedora, etc. - then you can probably just use your package manager and be up and running in a few minutes. For example, on most of my machines, I would open a terminal (Gnome Terminal, xterm, LXTerm, etc.) and type:

```bash
sudo apt-get update
sudo apt-get install git git-gui gitk
```

On an RPM-based distribution, I would do:

```bash
sudo yum install git git-gui gitk
```

You can also install from source, but usually that should be unnecessary unless your system is very out of date.

Notice those extra packages, `git-gui` and `gitk`? You might as well leave those off if you're running a text-only system (like a Raspberry Pi that you only access over a console cable), but on graphical desktops they're helpful tools for quickly committing work and reviewing history.
Installing Git on Windows

There are several approaches to installing Git on Windows. We'll start with the most basic, a port called msysGit which provides a version of the Bash shell that offers a bunch of Unix utilities along with the `git` commands.

What about GitHub for Windows?

GitHub offers a graphical desktop application for Windows which bundles both Git functionality and some of the features offered by GitHub.

I recommend against the use of this application, at least until you're familiar with the underlying tools. It's fairly opinionated about the proper way to use Git, which is of course fine (everybody has opinions), but it seems to obscure some important distinctions.

Installing msysGit

First, visit the official downloads page:

Click the "Windows" link, and run the installer.
Next, click your way through the installer wizard. You'll be asked a handful of questions. Generally, the defaults are safe.
Under "Select Components", check the box for an icon "On the Desktop".
The first option ("Use Git from Git Bash only") has the least impact on your system, but the second ("Use Git from the Windows Command Prompt") is fine too, and might make life a little easier if you're used to the Windows command line.

It's probably ok to leave line-endings as the default, but remember that line endings can have an effect on some projects! If you're going to share files with other operating systems, consider switching this to "Checkout as-is, commit Unix-style line endings".

Now you can watch the progress bar:
And now you should be able to find a desktop shortcut called "Git Bash" (unless you didn't check that option in the installation wizard, in which case you should look in the Start Menu, or press the Windows key and start typing "git").
Fire it up, and you should see a simple command prompt window:

Doesn't look like much, does it? This is actually a version of Bash, the standard shell on Linux systems.

Resources

- The Git Book: [Getting Started - Installing Git](#)
- Beanstalk Guides: [Working with Git on Windows](#)
Installing TortoiseGit on Windows

As an optional step for Windows users, you can install TortoiseGit (https://tortoisegit.org). msysGit already adds context (right click) menus for some Git features, but TortoiseGit provides a menu of nice GUI helpers for most important operations. It's not necessary, but it can show you some information quickly, and save some typing.

If you're not sure whether you want TortoiseGit, don't worry about it right now - you can always install it later!

First, visit tortoisegit.org (https://tortoisegit.org). This will redirect you to the project's current home page. (At the time of this writing, they're hosted by Google Code, but that will probably change in the near future.) Look for a "Download TortoiseGit" link, and from that page, get the appropriate version (32 or 64 bit) for your machine.
Run the installer, and click through the usual setup wizard. As with msysGit, default choices should be safe.
Click on the icons in the tree below to change the way features will be installed.

Click Install to begin the installation. If you want to review or change any of your installation settings, click Back. Click Cancel to exit the wizard.
Next, check that TortoiseGit is installed. Start by right-clicking on the desktop and making a new folder called something like "project" (you should already see some TortoiseGit options in the context menu):

Next, open the folder and right-click inside the window, then click "Git Create repository here...".
You'll get a couple of dialogs. Just press OK:

![Dialog 1](image1.png)

![Dialog 2](image2.png)

Now you should be able to right-click and see the full menu of TortoiseGit commands:
Don't worry if it looks complicated - most of the time you'll only use a handful of those commands.

**Installing Git on OS X**

Use [Homebrew](https://brew.sh). Visit their site for Homebrew installation details.

Once you have Homebrew, install like so:
Initializing a Repository and Making Commits

Initializing an Empty Repository

Now that you've got a working Git installed, let's create a test repository. On Linux or OS X, open a terminal (or switch to the one you used to install Git).

On Windows, open the Git shell by doubleclicking that icon on the desktop:
I like to keep all of my repositories in a "code" directory, so as not to clutter up the top level of my home directory or my desktop. You might want to do the same.

First, make an empty directory:

```
mkdir -p code/project
cd code/project
```

If you do `ls -a`, to list all files in the directory, you should just see `.` and `..`, special files which point at the current directory (`.`) and its parent (`..`).
Next, initialize an empty repository with `git init`, and try `ls -la` to show a long listing of all the files:

The `.git` directory you see is where Git will keep a record of changes you make to the project, along with metadata about things like where other copies of the repository live. (By longstanding convention on Unix systems, files that start with a leading dot are hidden from the user most of the time.)

For the most part, you don't need to worry about this directory, but it's useful to know that it exists. A Git repository is just like any other folder full of files, except that it contains a `.git`.

**Making a First Commit**

A commit, in the world of Git, is a collection of changes to files along with a (hopefully) human-readable description of the changes and some metadata about who and when. It's a little bit like an entry in a logbook.
In order to start recording changes to files, you'll first need at least one file to work with. By convention, most repositories have a README or README.md (formatted with Markdown ()) that describes their content. Let's make one of those:

```bash
touch README.md
lst
```

Now you'll want to add some text. On a Linux or OS X system, try nano README.md, write a description, and hit Ctrl-X to save:

```
@raspberryi ~code/project $ nano
```

On Windows, try typing start . to bring up an Explorer window for the current directory, and drag README.md into a text editor (Notepad will work just fine, in a pinch):
Once the file is saved, you can use `git status` to see what's changed:

```
@raspberrypi ~/code/project $ git status
# On branch master
# Initial commit
# Untracked files:
# (use "git add <file>..." to include in what will be committed)
# README.md
nothing added to commit but untracked files present (use "git add" to track)
```

Before we go any further, Git wants some basic information from us. Since every commit is associated with a user's name and e-mail address, let's set those system-wide:

```
git config --global user.email "youraddress@example.com"
git config --global user.name "Your Name"
```

Additionally, Windows users may want to set up a different editor for commit messages, since otherwise you'll find yourself using Vim, which can be profoundly confusing. Here's how to use Notepad:

```
git config --global core.editor "notepad.exe"
```

With that out of the way, here's how to commit the new file:

```
git add README.md
git commit
```

```
@raspberrypi ~/code/project $ git add README.md
```

`git add` is for adding (or "staging") files to something called the index, which is the list of changes that will be recorded in the next commit.
At this point, you should be looking at a text editor. Here is what you need to know:

- The first line is like the subject of an e-mail: A short description of the change you've just made.
- After the first line, you can add a blank line followed by a more detailed description of the change, if necessary. Sometimes people write many paragraphs here, depending on the scope of changes. Try to think like a future version of yourself who wants to remember what you were thinking at the time you did something.
- The first line should be short - 50 characters or so, if you can manage it.
- Subsequent paragraphs should be hard-wrapped (press enter) after 72 characters.
- Lines that start with a # are comments. They won't be recorded.

Something like "initial commit of README.md" is probably sufficient for a first commit like this one. Type that in, save the commit, and exit the editor (hit Ctrl-X in Nano, in other editors, make sure you save the file and then exit):

![Image of initial commit of README.md]

You should now find yourself back at the command prompt, looking at a summary of the commit something like this one:

```
pi@raspberrypi ~/code/project $ git add README.md
pi@raspberrypi ~/code/project $ git commit
[master (root-commit) d12928f] initial commit of README.md
 1 file changed, 4 insertions(+)
create mode 100644 README.md
pi@raspberrypi ~/code/project $  
```

Now you can use a couple of commands to see where you're at:

```
git status
git log
```
`git status` now lets you know that you haven't made any changes since the last time you committed.

`git log` displays the recent history of commits.

Notice the first line of that log entry? Yours will be different, but it will look a lot like this:

```
commit 813c3f799229bd1189e61514d9bb7f0e6620c30b
```

That long, not-especially-meaningful-looking string of numbers and letters is a hash, specifically a SHA-1() of various things(). This hash functions as a commit id, a unique identifier for each set of changes you record. From now on, you can always refer to that first commit by its hash.

You can also see the specifics of any given commit this way, with `git show <commit>`:

```
This will display not only the author, date, and full commit message, but also a unified diff() of the files that changed, with all of the changed lines.
```

You don't even have to use the full hash - the first 5 characters or so are usually unique. "git show 813c3" would do the trick here just as well.
Branches?

When running commands like `git status` or `git commit`, you might have noticed the text "On branch master" crop up repeatedly.

The basic idea of branching is that rather than a single stream of history, a repository can contain many, and different streams of history can be combined:

![Diagram of branching](image)

Although the specifics vary, modern version control systems generally support the concept of branches. In Git, branches are cheap, fast, and (usually) easy to work with, which has quickly made them an indispensable tool.

Once upon a time, when a programmer wanted to experiment with a different version of a program - for example, by adding a new feature - the safest thing was to make a copy of the files and work on the copy until it seemed to be good. Which is part of how you wind up with this scenario:
In Git, instead of making a direct copy of the files, you can say "I would like the history to fork here", like so:

```
$ git branch alternate_universe
$ git checkout alternate_universe
```

...and then you can go on your merry way, changing files and making commits like usual, and none of them will affect the other branches until (and only if) you’re ready to combine them, which you can do by saying:

```
$ git checkout master
$ git merge alternate_universe
```

Nothing says you have to merge any branches. If the changes I'm trying to make don't work out, I often simply discard the branch I've started like so:

```
$ git branch -d alternate_universe
```

...where `-d` is for "delete".

---

**Publishing Your Repository to GitHub**

Now that you know the basics of creating a repository and making commits, let's talk about how to share your work with other people.

This guide will focus on GitHub, but in fact there are lots of sites that will host a Git repository on the web for you - here's a [big list from the Git wiki](https://git.wiki). If you have a server sitting somewhere on the public web, you can provide your own interface with [Gitweb](https://gitweb) or a range of other tools. We use GitHub at Adafruit right now because it's both popular and a very good tool.

(Like a lot of things on the internet, this isn't a perfect state of affairs: there are reasons to be concerned that a single company controls the place where so much
open source code lives, and GitHub itself isn't open source, though they contribute a lot to open source projects. For now, I take the view that the perfect is the enemy of the good, and the distributed nature of Git makes this a lot less worrisome than it might be otherwise.)

Get You a GitHub Account

This part doesn't really need much explanation. Visit GitHub and pick a username and password (use a strong, unique password).

Once you're logged in, you'll be asked to choose a plan:

The free one is fine for most purposes - later you might want to upgrade, particularly if you need to host non-public repositories.

Choose a plan, and GitHub will drop you to its dashboard interface:
There's not much to see yet. That "Hello World" guide is pretty good, incidentally, but let's jump right into publishing the repo you've already started.

Create a Public Repository and Push Changes

Click "Create a repository", and you'll be given a simple form:

![Create a repository form](image)

Fill out the repository name and a short description. Don't initialize the repository with a README - you've already got one of those. Click "Create repository", and you'll be given some further instructions:
Make sure the "HTTPS" button is clicked (we'll talk about SSH later), and check out the bit under "...or push an existing repository from the command line". You want two commands:

```
git remote add origin https://github.com/yourusernamehere/project.git
git push -u origin master
```

The first tells Git to add a remote called "origin" at the specified address. A remote is just another repository (almost always a copy of the same one) that Git can push or pull changes to and from. You can verify this worked with `git remote -v`:

```
+ origin https://github.com/yourusernamehere/project.git (fetch)
+ origin https://github.com/yourusernamehere/project.git (push)
```

The second tells Git to push local changes on the branch called "master" to the remote called "origin". The `-u` option says "make master on origin an upstream", which means that from now on you can `git push` and `git pull` to GitHub and things will go to the right place.

You should be prompted for a username and password - use the ones you just created on GitHub:
Now reload the repository page in your web browser and you'll see something a lot like this:

Next time you make a commit, you can say `git push` without specifying a remote or a branch, and Git will use the new defaults.

HTTPS Credential Caching and SSH Keys

Cache Username and Password

You may have noticed that it seems a little tedious entering username and password every time you tell the `git` command to talk to GitHub.

One solution here is to configure Git to cache these credentials:

```
git config --global credential.helper cache
git config --global credential.helper 'cache --timeout=3600'
```
Generate an SSH Key

An even better idea is to generate an SSH key and add it to your account on GitHub. This will enable you to clone repositories from paths like so:

```
git@github.com:username/repo.git
```

Instead of:

```
https://github.com/username/repo.git
```

...and is almost certain to be more secure than authenticating with a password.

There's a nice GitHub guide on generating SSH keys (https://docs.github.com/resources/guides/), and their recommendations are, for the most part, fairly sound. Be aware that you'll probably need to change repositories you've already cloned to use the SSH URL (https://docs.github.com/resources/guides/ssl/) in order to take advantage of this.

GitHub documentation repeatedly recommends use of HTTPS URLs over SSH. This is very likely just because the HTTPS approach is less work for users and easier to explain.

Working With Existing Repositories

Cloning a Repository

A lot of the real utility of version control systems is that you can use them to access and keep track of other people's work.

If you look at Adafruit Industries on GitHub (https://github.com/adafruit), for example, you can find hundreds of repositories. Using the advanced search, you can see them sorted by how many people have starred them (https://docs.github.com/resources/guides/). (GitHub's equivalent of a bookmark or favorite.)

Adafruit-Raspberry-Pi-Python-Code (https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code) is one of the most popular of these, and a good fit for the Raspberry Pi we've been using for examples, so let's get a copy. Visit that page and look for an HTTPS clone URL:
It should be [https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code.git](https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code.git) - go back to your terminal and cd to your code directory, then clone the repo:

```
cd ~/code
git clone https://github.com/adafruit/Adafruit-Raspberry-Pi-Python-Code.git
```

Navigating History

Now check out the log of recent commits:

```
cd Adafruit-Raspberry-Pi-Python-Code
git log
```
If you want, you can scroll all the way through the history of the repository from here. You can also slice the display of commits in all sorts of ways - try `git help log` for a lot of detail. A couple of my favorites are:

```
I cheated before I ran that log command, and did "git config --global color.ui auto" on my Raspberry Pi so it would have nicer colors.

If you want, you can scroll all the way through the history of the repository from here. You can also slice the display of commits in all sorts of ways - try `git help log` for a lot of detail. A couple of my favorites are:

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```

```
The **--graph** option lets you see the flow of history as branches are created and merged back into master.

If you're working on a desktop system, you can probably even use `gitk`, which will give you an old-school but sometimes very useful way to navigate and search the project's history:

---

**Assigning Responsibility / History Within a File**

Among the most helpful questions a version control system can answer are these:

1. Who changed this line?
2. When?
3. Why did they change it?
4. What else did they change at the same time?

Suppose you're looking at a file, and you have one or more of these questions. Maybe you need to figure out who to talk to before you change a given piece of code, or make sure it wasn't your change that broke something. (It's amazing how often I have been working myself up to yelling at someone else about a bug only to check up and realize that it was my fault in the first place. Version control often breeds humility.)
There's a simple but powerful way to go about this. Let's say the file in question is `README.md`:

```bash
git blame README.md
```

This may look a little bit inscrutable at first glance, but it's a simple format. Here's the first line as an example:

<table>
<thead>
<tr>
<th>commit hash</th>
<th>author</th>
<th>date</th>
<th>line number &amp; text of line</th>
</tr>
</thead>
<tbody>
<tr>
<td>7b26f754</td>
<td>(Limor &quot;Ladyada&quot; Fried)</td>
<td>2012-08-17 19:58:16-0300</td>
<td>1) Adafruit's Raspberry-Pi Python Code Library</td>
</tr>
</tbody>
</table>

Now if you want to explore the changes that led to a file being in the shape it's in, you can use commands like `git show 7b26f754` to read the relevant commit message and see all the changes that were part of that commit at once.

This has a lot to do with why you should write good, thorough commit messages: They become a cheatsheet for a future version of yourself who has forgotten everything.
Submitting a Pull Request on GitHub

Once you know how to make commits, push to a GitHub-hosted remote, and clone a pre-existing repo, you're most of the way to submitting a pull request.

Pull requests are GitHub's way of modeling that you've made commits to a copy of a repository, and you'd like to have them incorporated in someone else's copy. Usually the way this works is like so:

1. Lady Ada publishes a repository of code to GitHub.
2. Brennen uses Lady Ada's repo, and decides to fix a bug or add a feature.
3. Brennen forks the repo, which means copying it to his GitHub account, and clones that fork to his computer.
4. Brennen changes his copy of the repo, makes commits, and pushes them up to GitHub.
5. Brennen submits a pull request to the original repo, which includes a human-readable description of the changes.
6. Lady Ada decides whether or not to merge the changes into her copy.

Let's walk through a basic pull request using a very simple repo called Adafruit-Git-Intro.

Get a Copy to Work On

First, visit https://github.com/adafruit/Adafruit-Git-Intro:

Notice that little "Fork" button up in the corner? Click that:

You might be asked where you want to fork the repo. In this case, I'm forking it to my personal account instead of to the account for any of the organizations I'm a member of:
...and here's my personal copy (), which can be cloned from

```
$ git clone
```

### Make Some Changes

As an example, I'm going to make a pull request for a file called CONTRIBUTORS.md, with a list of people who have contributed to the repo. I'll add my name to the top - when making your own pull request, you can add yours to the list.

```
nano contributors.md
```
Add a line with your name and hit Ctrl-X to exit and save, and check the status:

```
@raspberrypi ~/code/Adafruit-Git-Intro $ git status
On branch master
Your branch is up-to-date with 'origin/master'.
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git checkout -- <file>..." to discard changes in working directory)

modified:  contributors.md
```

no changes added to commit (use "git add" and/or "git commit -a")

```
@raspberrypi ~/code/Adafruit-Git-Intro $
```

Now make a commit:

```
git add contributors.md

git commit
```

```
GNU nano 2.2.6 File: .../nen/code/Adafruit-Git-Intro/.git/COMM\nIT_EDITMSG Modified
add Your Name to contributors.md
# Please enter the commit message for your changes. Lines starting
# with '#' will be ignored, and an empty message aborts the commit.
# On branch master
# Your branch is up-to-date with 'origin/master'.
# Changes to be committed:
#     modified: contributors.md
```

```
@raspberrypi ~/code/Adafruit-Git-Intro $ git commit
[master a7720fa] add Your Name to contributors.md
  1 file changed, 4 insertions(+)

git@raspberrypi ~/code/Adafruit-Git-Intro $
```

...and push your new commit back up to your copy of the repo on GitHub:
Submit the Pull Request

Now have a look at the repo on your GitHub account. You should see a notification that your master branch is 1 commit ahead of adafruit:master (here, mine is actually a couple ahead, since I've been messing with some other things). Look to the right side of that notice, and you'll see a "Pull Request" link. Click it.

Now you should be looking at a comparison of your repo and the official Adafruit copy, with your commits and additions shown:

Go ahead and click the big green "Create Pull Request" button. You'll get a form with space for a title and longer description:
Like most text inputs on GitHub, the description can be written in [GitHub Flavored Markdown](https://guides.github.com/features/mastering-markdown/). Fill it out with a description of your changes. If you especially want a user’s attention in the pull request, you can use the "@username" syntax to mention them (just like on Twitter).

GitHub has a handy guide to [writing the perfect pull request](https://guides.github.com/pull-requests/) that you may want to read before submitting work to other repositories, but for now a description like the one I wrote should be ok. You can [see my example pull request here](https://github.com/Adafruit/Adafruit/blob/master/Adafruit_ASKR_DSK012/ASKR_DSK012.ino#L33).

That's pretty much it! From here, the owner of the repository (in this case, the people with access to Adafruit’s GitHub account) can comment on individual changes or the entire pull request, and choose to accept or reject the request. They may have questions for you, suggestions for improving your changes, or feedback on your overall goals.