



Contribute to CircuitPython with Git and GitHub

Created by Kattni Rembor



<https://learn.adafruit.com/contribute-to-circuitpython-with-git-and-github>

Last updated on 2025-03-22 10:50:24 AM EDT

Table of Contents

Overview	5
<ul style="list-style-type: none">• Requirements• Expectations	
Grab Your Fork	7
<ul style="list-style-type: none">• Clone Your Repo	
Always Work on a Branch	11
<ul style="list-style-type: none">• Starting from the Right Place• Main or something else?• Updating the Main Branch• Create Your New Branch• Moving your Local Clone and Fork to the main default branch	
Check your code with pre-commit	17
<ul style="list-style-type: none">• One-time initial install of pre-commit• Workaround for pre-commit issues on Ubuntu 22.04 and Debian• Per-repository installation• Running pre-commit• More Info	
Status, Add, Commit, Push	19
<ul style="list-style-type: none">• Branched and Ready to Code• git status is Your Best Friend• Time to commit• Second commit and Further• push to Your Fork• My push Failed!• CircuitPython: pushing to the main Branch?	
GitHub Personal Access Token	25
<ul style="list-style-type: none">• A Note about GitHub Personal Access Token Types• The Personal Access Tokens Menu• Generating a New GitHub Token (classic)• Your New Token• Authenticating via Command Line• Further Information	
Add Author and License Information	32
<ul style="list-style-type: none">• Arduino• CircuitPython• SPDX• Troubleshooting	
Create Your Pull Request	34
<ul style="list-style-type: none">• Creating a Pull Request from Your push• Creating a Pull Request through GitHub• Creating a PR from the Original Repository• Creating a PR from Your Fork• Open a Pull Request• Automatic Merge Verification• Pull Request Title and Message	

- Commit List
- Your Changes
- Create Your Pull Request

Your Open Pull Request 40

- Your Pull Request Explored
- Header
- Comment
- Commit List
- Status and Merge Info
- Leave a Comment
- The Right Column
- The Files Changed Tab
- Time for Action!
- Some Checks Haven't Completed Yet
- All Checks Have Failed
- Pushing the Fix to Your PR
- All Checks Have Passed

Receiving a Review 50

- The Change Request
- The Conversation View
- The Files Changed View
- Submitting the Requested Changes
- Using Git to Commit Suggested Changes
- Using GitHub to Commit Suggested Changes
- Don't Let Your Local Code Get Behind
- Changes Approved
- Merge
- Post-Merge Cleanup

Staying Up To Date 61

Keeping Branches Trimmed 63

- Deleting Your Remote Branch
- Deleting Your Local Branch

Giving a Review 66

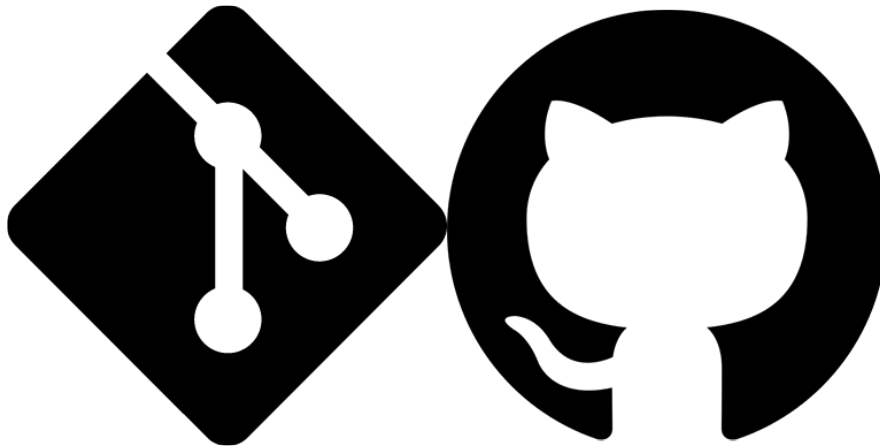
- A Positive Experience
- Someone Opened a Pull Request
- Begin Your Review
- Review Response and Update
- Review Merge

Starting Over Fresh 79

- Deleting Your Local Clone
- Deleting Your Fork
- Forking Again
- Cloning Again

Glossary 83

Overview



This guide has been updated due to an update to GitHub. The major steps in this guide now refer to different libraries. It is still straightforward to follow along, as the concepts here apply to any library.

Just about all of Adafruit's code and hardware is kept on GitHub - a web service that keeps track of code and files. Since we publish open source hardware and software, this works great to share our designs and also get feedback and improvements from the community. By working together, a large group of people can improve and build upon the body of work that Adafruit has published. You can even find bugs or add new features, and submit those back to us so that everyone can benefit from your effort!

But how do you actually do that? GitHub isn't the easiest site to use, and Git the versioning tool it builds upon can be challenging even for coding experts.

This guide aims to not only show you where to start, but provide you with the entire contribution path, beginning to end. I'll be using CircuitPython as our example as I have a well established workflow.

This guide assumes you already have a [GitHub account \(https://adafru.it/d6C\)](https://adafru.it/d6C) and have installed Git. You're ready to contribute code. Perhaps you've found an issue with CircuitPython library. It doesn't currently work properly, and you think you know how to fix it! Now, where do you start?

This guide will walk through all of the steps I follow during the contribution process. You'll learn how to fork and clone a project repository, create a working branch, and commit and push your changes. You'll find out how to create a pull request, and progress through the review process including the conversation and work

surrounding a change request. I also explain what's involved with giving a review, which is another excellent way you can contribute to a project.

Some of this may sound complicated and confusing. The guide intends to change that. However, if you find that you're still confused while going through the guide - don't worry! GitHub is hard to explain, and if you have suggestions on how to improve this guide, please feel free to tell us where things get confusing or could be clearer. You can click the feedback link found on the left of the guide, or find us [Discord \(\)](#) in the #circuitpython channel. We'd love to hear from you!

All of the terms introduced in this guide are explained as you are introduced to them, and are also defined in the [Glossary found at the end of the guide \(https://adafru.it/B10\)](https://adafru.it/B10). If you're ever unsure about a term, feel free to look it up there.

First, you need to make sure you're ready to get started with this guide.

Requirements

Before starting this guide, there are a number of steps found in [An Introduction to Collaborating with Version Control \(https://adafru.it/BHW\)](https://adafru.it/BHW) that you must complete. You must have [Git installed and setup on your computer \(https://adafru.it/B13\)](https://adafru.it/B13). You must have a [GitHub \(https://adafru.it/d6C\)](https://adafru.it/d6C) account. You need to have some familiarity with the command line or be ready to learn. This guide uses a terminal program to interact with Git locally (on your computer).

Expectations

Be aware that this guide has a very specific goal. It is designed to provide you with a complete open source project contribution workflow, beginning with forking the original project GitHub repository.

This guide will not help you with your initial setup of Git or creating your GitHub account. There is no information about troubleshooting issues with your Git setup or configuration. It will not explain how Git and GitHub work. Much of this is covered in [An Introduction to Collaborating with Version Control \(https://adafru.it/BHW\)](https://adafru.it/BHW). Further information is likely available through the [Git documentation \(https://adafru.it/B11\)](https://adafru.it/B11) or the [GitHub documentation \(https://adafru.it/B12\)](https://adafru.it/B12).

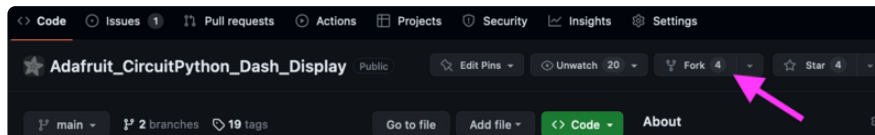
Though I primarily work with CircuitPython and related projects, the workflow outlined in this guide should apply to any open source project. However, it's always a good idea to check with the project's maintainers to make sure that you're working within their guidelines.

Let's get started!

Grab Your Fork

When you're contributing to a project, you typically don't edit the project directly. You create a copy of the project's **repository**, or **repo**, for yourself and make all of your changes there before submitting them to the project. This copy is called a **fork**.

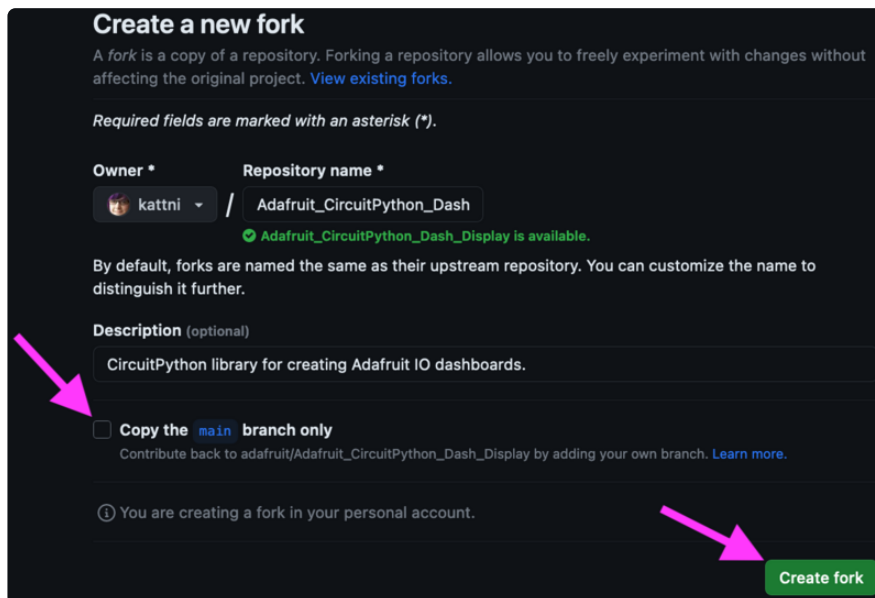
To begin, you must be signed into your GitHub account. Then use a browser to navigate to the repo for the project to which you plan to contribute. I'm going to be contributing to the Adafruit CircuitPython Dash Display library.



The first thing you want to do is fork the repo. Click the **Fork** button on the right side of the page to fork a copy of the repo to your account.

In the Create a new fork dialogue, uncheck the "Copy the **main** branch only" box to get all the branches of the repo.

Finally, click **Create fork** to continue.



The **Create fork** button will change text for a moment before you see the following.

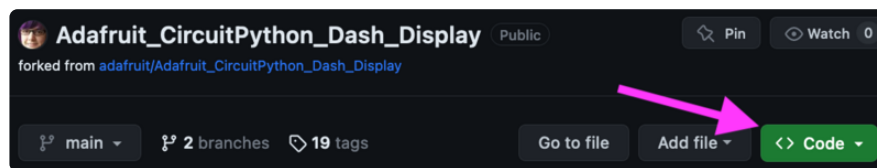


Great job! You now have your very own GitHub copy of the project repo to which you're going to contribute. You're ready for the next step!

Clone Your Repo

The next thing you'll need to do is download a copy of your new repo on your computer so you can start working on it. This is called **creating a clone** or **cloning**. Create a directory on your computer to hold your projects. Mine is in my home folder and is called **repos**.

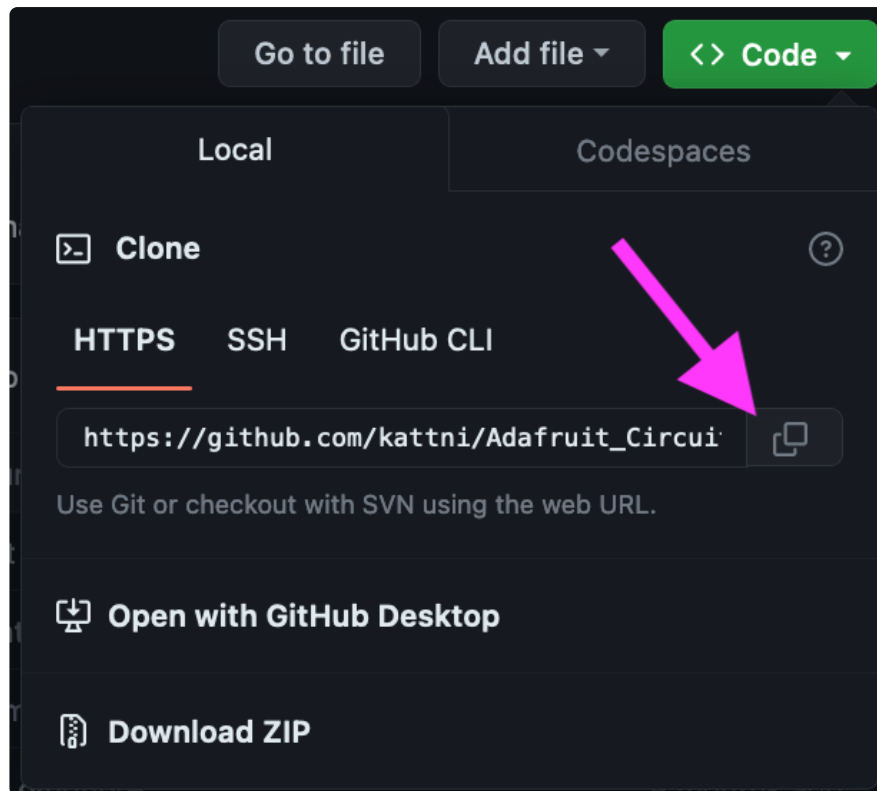
On the page for your repo, you'll find a green **Code** button.



Click it to find the clone URL for your new fork. The URL will look something like this:

https://github.com/kattni/Adafruit_CircuitPython_Dash_Display.git (<https://adafru.it/19dV>)

Click the copy button (the two-box icon to the left of the URL) to copy the URL to your clipboard.



Open your terminal program and navigate to your new directory using the `cd` command.

```
kattni:~  
> cd repos
```

When you clone a repo, Git assigns the repo on GitHub an alias, which by default is "origin". You may in the future have a reason to clone a repo that is not your own fork, and it can be confusing when all repos are called `origin`. So, for my repos, I like to set the remote alias to the repo owner's GitHub user ID. In the case of my fork, this is my GitHub user ID. This makes it easier to remember when I'm contributing to my own repos versus contributing to someone else's repo.

Once you're in your new directory, enter the `clone` command. Replace `remotename` with your GitHub user ID, and paste the URL from your clipboard:

```
git clone -o remotename https://your-fork-URL
```

```
kattni:~/repos (code-checks) v3.11.3  
> git clone -o kattni https://github.com/kattni/Adafruit_CircuitPython_Dash_Display.git  
Cloning into 'Adafruit_CircuitPython_Dash_Display'...  
remote: Enumerating objects: 378, done.  
remote: Counting objects: 100% (26/26), done.  
remote: Compressing objects: 100% (23/23), done.  
remote: Total 378 (delta 9), reused 11 (delta 3), pack-reused 352  
Receiving objects: 100% (378/378), 93.36 KiB | 1.61 MiB/s, done.  
Resolving deltas: 100% (223/223), done.
```

This will create a local copy of the repository on your computer in a directory with the same name as the repo. So, now I have a new directory, `~/repos/`

`Adafruit_CircuitPython_Dash_Display`, which contains a copy of the newly forked repo.

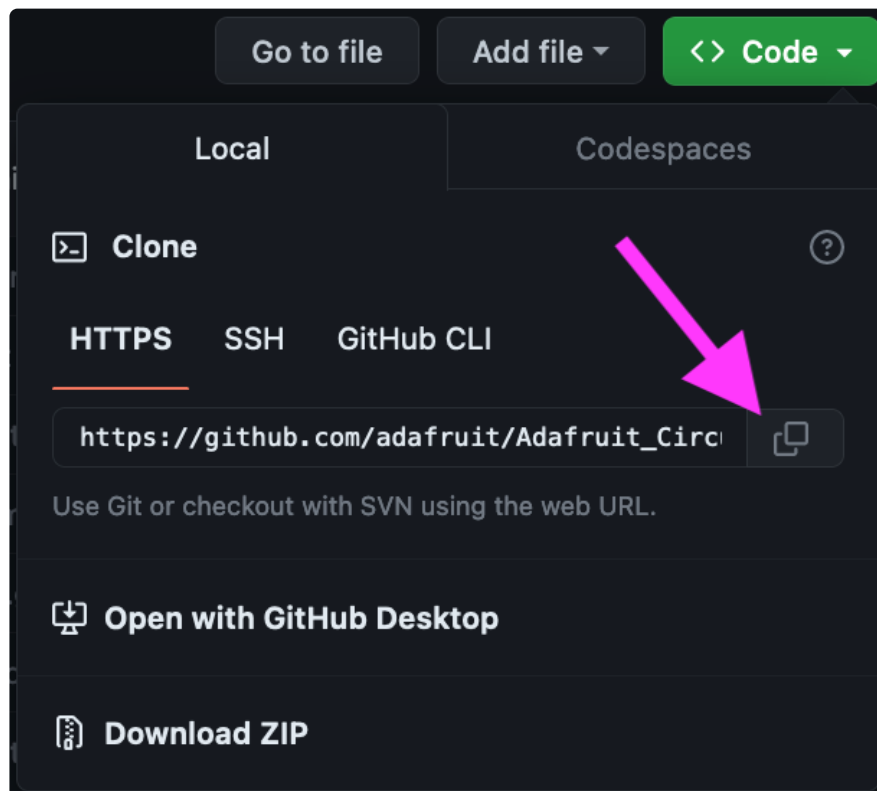
Now, use the `cd` command to move into that directory.

```
kattni:~/repos (code-checks) v3.11.3  
> cd Adafruit_CircuitPython_Dash_Display
```

When changes are merged with the project, they're merged into the original repo. This includes your final changes and changes submitted by others.

Changes made to the original repo do not automatically update to your repo. You have to manually fetch those changes and merge them into your own repo. To do this, you need to add what is called a **remote**. A remote allows you to fetch the changes from the original project to stay updated.

Use your browser to navigate to the page for the original repo. Click the **Code** button, then click the **copy** button to copy the URL for the original repo to your clipboard. Remember to use SSH if you setup GitHub to use it.



When you create a copy of a project, the original project is considered to be **upstream**. Since you're getting the updates from upstream, often the remote is called **upstream**. However, in exactly the same way we called your remote your GitHub ID, it's easier to call the original remote by the owner's GitHub ID. The original repo here is owned by Adafruit, so I'm going to name the remote **adafruit**.

While in the directory for your newly cloned repo, enter the following **remote** command, changing **remotename** to the original project owner's GitHub ID, and pasting the URL from the clipboard:

```
git remote add remotename https://original-project-url
```

```
> git remote add adafruit https://github.com/adafruit/Adafruit_CircuitPython_Dash_Display.git
```

Your local repo is set up and ready to go. You forked the repo, cloned a local copy, and prepared to keep it updated.

Now you're ready to begin working with it!

Always Work on a Branch

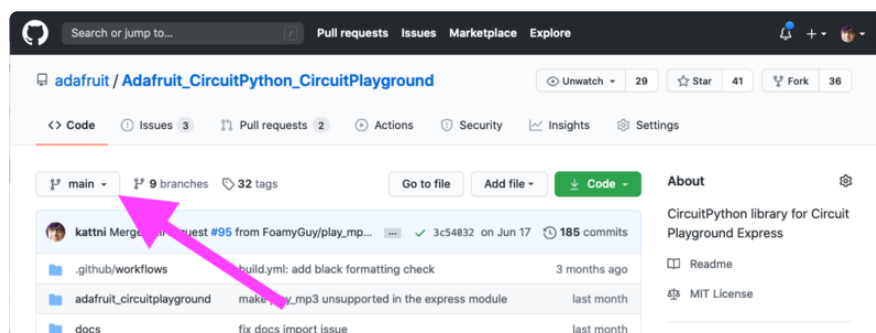
Now that your local repo is set up and ready to go, it's time to start working with it.

Starting from the Right Place

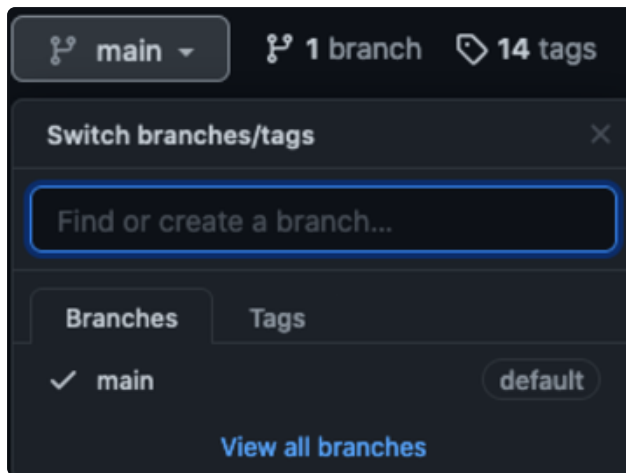
Imagine you made a change to your code, but made a mistake. Now your repo is in a bad state. To help avoid this situation, we use branches. You always want to make changes while on a **branch**. A **branch** is a way to have your own working timeline of changes, while leaving the default branch even with the original project. The default branch is called **main**. It's best to leave main clean, and make your changes on a working branch. For more details about branches, [check out the Branches? page \(https://adafru.it/BI4\)](https://adafru.it/BI4) found in the Adafruit guide [An Introduction to Collaborating with Version Control \(https://adafru.it/BIj\)](https://adafru.it/BIj).

Main or something else?

You may want to determine whether the library you are working with is using **main** or something else as the default branch. This is a simple process. First, visit the library on GitHub. Above the repo contents on the left side is a drop down menu that shows all available branches.



It will typically be on the default branch to begin with, but you can verify by clicking on the menu.



The default branch will be will be the only branch in the list with "default" next to it.

Updating the Main Branch

If you just cloned your repo for the first time, you're using the most up-to-date version as your start point. However, if you cloned it a while ago, or this is not your first time contributing, you may not be up to date. So, before you begin, you want to make sure the main branch is current.

To create a new branch or move between existing branches, you'll **checkout** the branch you'd like to switch to. The checkout command allows you to switch to a new branch, by creating it in the process, or to switch to an existing branch.

To update main, first checkout main to verify you're on the correct branch:

```
git checkout main
```

```
1486 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git checkout main
Switched to branch 'main'
Your branch is up to date with 'kattni/main'.
```

Next, we're going to utilise the original project remote we created. To get the updates from the remote repo, we're going to use **fetch**. **fetch** grabs the the newest version of the remote repo, but does not merge it into the current repo.

Remember, you named the original project's remote repo with the owner's GitHub ID. You'll use this name when you merge the two main branches together. Since I cloned an Adafruit repo, I'll be using **adafruit**.

To fetch the updated remote, enter the following **fetch** command, replacing **ownerID** with the name you assigned to the original project's remote repo:

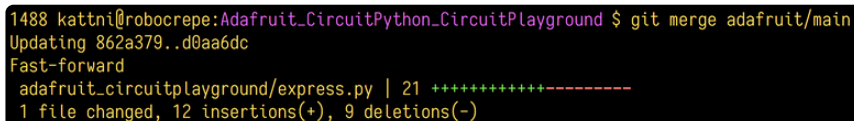
```
git fetch ownerID
```

```
1487 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git fetch adafruit
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From github.com:adafruit/Adafruit_CircuitPython_CircuitPlayground
 862a379..d0aa6dc  main    -> adafruit/main
```

Now we're going to **merge** the current data into our local repo. A **merge** takes the information from one branch and combines it into another. In this case, it's going to take the current version of main from the remote repo and combine it with the main branch on your local repo. This will bring you even with the remote main, and that means you're up to date.

To merge the remote main with your main, run the following **merge** command, replacing **ownerID** with the name you assigned to the original project's remote repo:

```
git merge ownerID/main
```

A terminal window with a dark background and light-colored text. The prompt is '1488 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground \$'. The command entered is 'git merge adafruit/main'. The output shows 'Updating 862a379..d0aa6dc', 'Fast-forward', a diff for 'adafruit_circuitplayground/express.py' with 21 lines of changes (12 insertions, 9 deletions), and a summary '1 file changed, 12 insertions(+), 9 deletions(-)'.

```
1488 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git merge adafruit/main
Updating 862a379..d0aa6dc
Fast-forward
 adafruit_circuitplayground/express.py | 21 ++++++++-----
 1 file changed, 12 insertions(+), 9 deletions(-)
```

There have been some updates to the remote main since I last did anything with this repo. Good thing I updated!

Now your main branch is even with the original project's main branch and you're ready to create your working branch!

Alternatively, you can simply run **git checkout ownerID/main** (where **ownerID** is the name you assigned the original project's remote repo) and then continue with the next set of steps. It will not update your main branch, but it will ensure that you create your new branch from the most updated version of the repo.

Create Your New Branch

Now we can create a new branch. It's good practice to create a new branch for each new contribution you are working on. I'm working on fixing a bug with audio, so I'll be doing all of it in one branch. However, if I intended to submit a fix for audio and another one for adding a new function to the library, I would want to work on one and then the other in two separate branches. This helps keep reviews simpler and more effective by delineating separate concepts and allowing you and the reviewer to focus on each one properly.

You can name a branch whatever you'd like, however, it's useful to name the branch something descriptive of the work that will be going on within it. I'm going to be submitting a fix to the **play_file** and **stop_tone** functions of the Circuit Playground Express library. So, I'm going to name my branch **play-file-stop-tone-fix**.

To create a branch, enter the following **checkout** command, replacing **your-branch-name** with whatever you'd like to call your branch:

```
git checkout -b your-branch-name
```

```
$ git checkout -b play-file-stop-tone-fix  
Switched to a new branch 'play-file-stop-tone-fix'
```

If you've already created a branch and you'd like to return to it, you can enter:

```
git checkout your-branch-name
```

If you'd like to return to the main branch, you can enter:

```
git checkout main
```

Now that you've created your branch, it's time to get to work!

Moving your Local Clone and Fork to the main default branch

These steps are only necessary if you forked an Adafruit repo BEFORE it was moved to main. If you forked the repo AFTER the move to main, your default should already be main.

Once the original Adafruit GitHub repository has moved to main, you will want to update your fork to have main as the default branch as well.

I recommend following the steps found on the [Starting Over Fresh page \(https://adafru.it/TcN\)](https://adafru.it/TcN) in this guide to update your fork and local clone.

However, if you have local work you want to keep, or you'd rather do it manually, you can run the following commands from command line (in your terminal window, inside the repo directory), replacing `YourUserID` with the name of your remote:

```
git pull adafruit  
git checkout main  
git push YourUserID main
```

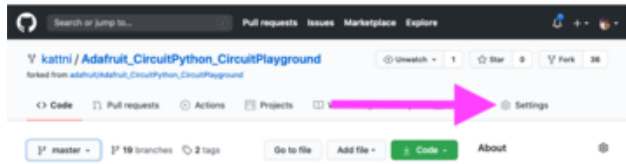
Once you've pushed the main branch to your remote fork, you should delete the previous default branch, by running the following command. This will eliminate any chance of you trying to use the wrong branch.

```
git branch -d main
```

Next, go to your fork on GitHub. The URL of your fork is the same as the Adafruit repo URL, except with your user ID in place of Adafruit. For example, my fork of the Circuit Playground library is found at the following URL:

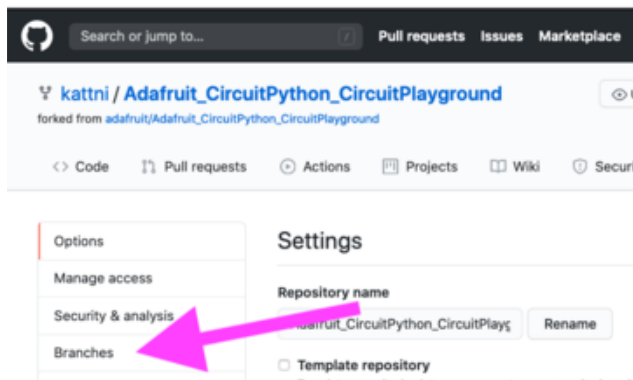
https://github.com/kattni/Adafruit_CircuitPython_CircuitPlayground (<https://adafru.it/MeN>)

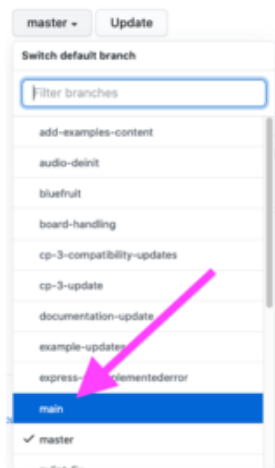
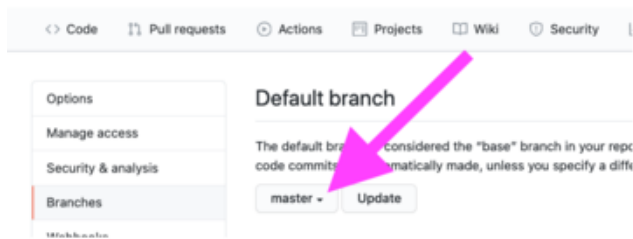
Once there, complete the following steps:



Click **Settings**, the right-most tab below the repo name.

Once in Settings, click **Branches** on the left.

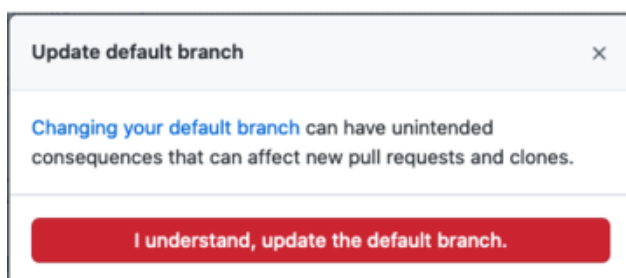


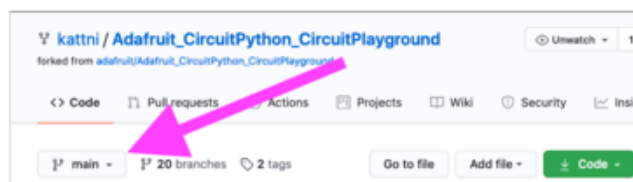
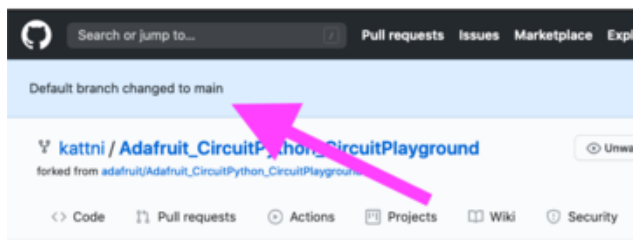


Click the drop down below Default Branch.
Choose **main**.
Click **Update**.
Click "I understand, update the default branch."

Default branch

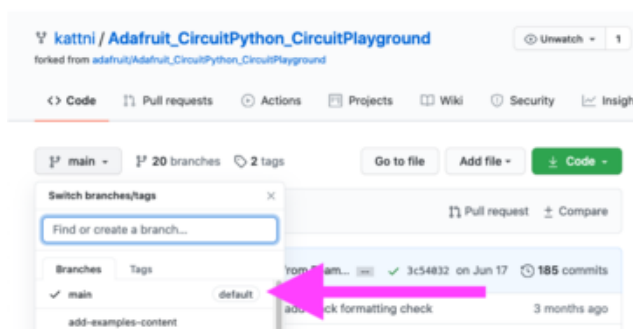
The default branch is considered the "base" branch in your repository, against which all pull requests and code commits are automatically made, unless you specify a different branch.





A notification will appear above the repo name that says "**Default branch changed to main**".

If you like, you can now verify that the default branch has been changed by clicking on the branch list drop down, and verifying that it says "**default**" next to "main".



You've now successfully updated your fork to the main default branch. You're ready to continue!

Check your code with pre-commit

Maintaining code is quite a challenge over the long-term. Over the lifetime of a library, many people read and edit it. Each person has a different background in coding and also a different goal in mind. These variations can lead to inconsistencies throughout

the code that makes it a bit harder for the next person to understand. In CircuitPython libraries, we use tools like Pylint and Black to ensure consistency in new code.

As we've added more automated checks, we've changed to a system called **pre-commit** to manage the checks overall. Once installed properly, you can run pre-commit locally, before committing new code into Git. It also runs remotely on GitHub when a pull request has been proposed. pre-commit is set up to remotely run on all existing libraries. It will automatically run remotely on a new library thanks to cookiecutter.

However, **it won't run locally unless you install it into your local directory**. We highly recommend doing this because it will both check and fix your code locally.

One-time initial install of pre-commit

If you've never used **pre-commit** on your computer before, you'll need to install it **globally** (there is a second "install" for each repository.) The easiest way to install it is with **pip**.

```
pip install pre-commit
```

Workaround for pre-commit issues on Ubuntu 22.04 and Debian

In ubuntu 22.04 or the analogous Debian release, you may see the error "expected environment for python to be healthy immediately after install" when trying to use **pre-commit**. To fix this, add this line to your **.bashrc** or **.bash_aliases** file, or other shell startup file. Restart your shell as necessary to pick up this setting.

```
export SETUPTOOLS_USE_DISTUTILS=stdlib
```

This **export** must be present before **pre-commit** sets up its **virtualenv** environment, which happens the first time you do **pre-commit run** or you try to push a commit. If the **virtualenv** is already set up, do **pre-commit clean**, which removes the existing **virtualenv**.

See the instructions from [the pre-commit project for installation](#) for alternative ways of installing **pre-commit**.

Do not use the `export SETUPTOOLS_USE_DISTUTILS=stdlib` on Ubuntu 24.04 or Debian Bookworm. It will cause some uses of `pip install` to fail.

Per-repository installation

For **every new repository**, you'll need to perform a pre-commit installation. This installs the specific versions of checks that the repository specifies. From within the repository do:

```
pre-commit install
```

After running this command, pre-commit will automatically run when you do `git commit`.

However, if you don't do this, you can still run pre-commit manually.

Running `pre-commit`

`pre-commit` will run each check every commit for all of the modified files and either pass or fail. Most checks that fail will also modify the source file to make it pass (like removing extra spaces). Once that happens, you'll see newly modified files in `git status`. `git add` them and then try the commit again.

Manually

You can run the pre-commit checks on every file whenever you like with:

```
pre-commit run --all-files
```

More Info

For more info on `pre-commit` see pre-commit.com.

Status, Add, Commit, Push

Branched and Ready to Code

If you're planning to edit a currently existing file, open that file, from within your local copy of the repo, into an appropriate editor and make your changes. If you're planning on adding a new file, create, edit, and save that file into the correct directory

inside your local copy of the repo. Once you've made a set of changes, it's time to **commit**.

A **commit** is a save point in your project. It's similar to saving a file to your computer, however, instead of overwriting the previous save, it creates a timeline of save points. You can return to a previous save point at any time.

To best be able to utilise commits, you need to make them often. Lots of little commits creates many "undo" points in your project. This way, if you head down the wrong track or find your changes aren't working, you can easily return to the last known-good point and work from there.

As well, you can use committing often to divide up your set of changes. Consider a commit to be a complete and distinct idea. Each time you complete a concept you wanted to change, commit. The sum of these commits will be a combination of all the changes you intend to submit to the final project. This creates a timeline for your set of changes and allows for a better understanding of what your train of thought was while you were completing them. This can make it easier for you to make changes later, and easier for a reviewer to see where you were going with your ideas.

The first thing you want to do when you're ready to commit, is check the **status**.

git status is Your Best Friend

When inside your repo, before you run any commands, you always want to run **git status**. This provides you with the state of your changes. Knowing the current status can help you know what command to run next. For example, If you have **Changes not staged for commit:** the next command you may want to run is **git add** to add your changes to be committed. If you have **Changes to be committed:**, the next thing you may want to do is run **git commit** to commit your changes. Don't worry, we'll cover all of this!

The important thing is to run **git status** every time before you run anything else so you know where you are.

Time to **commit**

It's time for your first commit. The first thing you'll do is run **git status**.

```
1449 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground$ git status
On branch play-file-stop-tone-fix
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:   adafruit_circuitplayground/express.py

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

As you can see, I've modified the `express.py` file. It's listed under **Changes not staged for commit:**. Before I go any further, I'd like to make sure I've made all of the changes I intended to. So, I'm going to run `git diff`.

`git diff` compares two states of the file. The first state is the original state if this is your first time editing it or the state since the last commit if you've already made a series of commits. The second state is the current state including your changes. It provides a color coded look at the difference between the two files, which highlights all the changes you've made. It only shows you the code near your changes - some files are extremely large and it would take forever to scroll through the entire file to look at a small change. Be aware, there are times when you'll make many changes, and the results of `git diff` will take a long time to go through.

To see your changes, enter the `git diff` command.

```
1443 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git diff
diff --git i/adafruit_circuitplayground/express.py w/adafruit_circuitplayground/express.py
index ef90638..e76af1b 100755
--- i/adafruit_circuitplayground/express.py
+++ w/adafruit_circuitplayground/express.py
@@ -649,6 +649,8 @@ class Express:    # pylint: disable=too-many-public-methods
     # Stop playing any tones.
     if self._sample is not None and self._sample.playing:
         self._sample.stop()
+        self._sample.deinit()
+        self._sample = None
     self._speaker_enable.value = False

     def play_file(self, file_name):
@@ -677,6 +679,7 @@ class Express:    # pylint: disable=too-many-public-methods
         audio.play(file)
         while audio.playing:
             pass
+        audio.deinit()
     else:
         audio = audioio.AudioOut(board.SPEAKER, open(file_name, "rb"))
         audio.play()

1443 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

I've added three lines of code. These are the green lines denoted with a plus sign at the beginning of the line. These are all the changes I'd like to make for now. So I'm certain I'm ready commit.

It's always a good idea to run `git status`.

```
1444 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git status
On branch play-file-stop-tone-fix
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:   adafruit_circuitplayground/express.py

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

Remember, my file is still listed as **Changes not staged for commit:**. This means before I can commit it, I must use `git add`.

To prepare a changed file to be committed, you must run `git add`. `git add` adds the file to the list of files to be committed. You can add as many changed files as you like to that list.

There are two ways to add files for commit. You can **add individual files** by using the file name:

```
git add adafruit_circuitplayground/express.py
```

Or, you can **add all files not staged for commit** by running the following:

```
git add .
```

This is super helpful when you've modified multiple files, and would like to add them all at once!

To add your file to the list, enter the `git add` command:

```
1451 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git add adafruit_circuitplayground/express.py
```

Followed by, you guessed it, `git status`:

```
1452 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git status
On branch play-file-stop-tone-fix
Changes to be committed:
  (use "git reset HEAD <file>..." to unstage)

    modified:   adafruit_circuitplayground/express.py
```

You'll see that you now have **Changes to be committed:**. Any files under this list will be added to the current commit. The only file I have listed is `express.py` because that's the only file I've changed. Since that's the only file I'm planning to add, I'm ready to commit.

When you commit, you'll enter a commit message. This message is a short description of the change you're committing. It should be 72 characters or less. If you're committing a new file for the first time, it's common practice to use the commit message, **"Initial commit."**. Otherwise, it can be whatever you like.

To commit your file, enter the following command, replacing **Commit message** with your commit message:

```
git commit -m "Commit message"
```

```
1453 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git commit -m "Added deinit() to play_file, stop_tone"
[play-file-stop-tone-fix 13c4623] Added deinit() to play_file, stop_tone
1 file changed, 3 insertions(+)
```

If you made a significant number of changes, you may want to leave a longer commit message. You'll want to setup Git to use your editor of choice by following the

instructions found [here \(https://adafru.it/BHY\)](https://adafru.it/BHY). Typically it defaults to vim. Windows users will probably want to check [here \(https://adafru.it/BHY\)](https://adafru.it/BHY) to set the editor to notepad, notepad++, etc. unless you really want to use vim.

Second `commit` and Further

That was the first change I wanted to make. Remember, it's good practice to commit each time you complete an idea or concept. This change was a complete concept for me, so I committed.

However, there's another issue with the library that I need to resolve as well. So, I'm going to add those changes, and follow the steps again. I make my changes, check the `status`, check the `diff` to make sure I made the correct changes, `add` the file to be committed, compose a short commit message, and `commit` my changes.

You can repeat the steps above as many times as you'd like.

Once you've committed all of the changes you intend to make, you're ready to push to your fork.

`push` to Your Fork

You've committed your final change, and you're ready to submit your code to the project. This means it's time to `push` to your fork. When you `push`, you're sending the list of commits since the last push to your remote repo. In other words, you're "uploading" your changes to your repo on GitHub. Until you `push`, none of your commits show up on GitHub. So think of commits as local save points, and pushes as remote save points. This also means that once you `push`, your changes are visible to the public. So `commit` as often as you like, but only `push` when you're ready for it to be submitted to the project. If you do `push` too soon, it's okay though! It happens to all of us. You can always `push` again after you do a few more commits.

As usual, first run `git status`.

```
1469 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git status
On branch play-file-stop-tone-fix
nothing to commit, working tree clean
```

When `status` results in `nothing to commit, working tree clean`, it means there have been no changes to any files in your repo since the last time you committed. This is the state you want to be in before pushing your changes.

Now, you want to enter the `push` command. Remember, when we setup the repo, we aliased it to your GitHub ID so you'd know it's your repo. The `push` command consists of the command, your alias, and your branch name. So, enter the following,

replacing `yourid` with your GitHub ID, and `your-branch-name` with the name of your branch:

```
git push yourid your-branch-name
```

```
> git push kattni f-strings-and-params
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 12 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 752 bytes | 752.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
remote:
remote: Create a pull request for 'f-strings-and-params' on GitHub by visiting:
remote:   https://github.com/kattni/Adafruit_CircuitPython_Dash_Display/pull/new/f-strings-and-params
remote:
To github.com:kattni/Adafruit_CircuitPython_Dash_Display.git
 * [new branch]      f-strings-and-params -> f-strings-and-params
```

Excellent! Now you can continue working. Or if you're ready, you can head over to GitHub to prepare to open a pull request.

My `push` Failed!

If this is your first time going through this process, you will likely be asked to enter your GitHub username and password. The intuitive thing to do is to enter the username and password you use to login to GitHub. However, you'll find that this will not work. You'll receive a warning, and a `fatal: Authentication failed` error.

```
> git push kattni test
Username for 'https://github.com': kattni
Password for 'https://kattni@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com:kattni/Adafruit_CircuitPython_NeoPx18.git/'
```

Don't despair! There is a [GitHub URL \(https://adafru.it/19dX\)](https://adafru.it/19dX) provided in the warning which takes you to their documentation. However, this particular page is not necessarily geared towards beginners, and requires you to dig deeper to find the info you need. Of course, this guide has you covered! It's time to learn how to generate a **personal access token**.

CircuitPython: `push` ing to the `main` Branch?

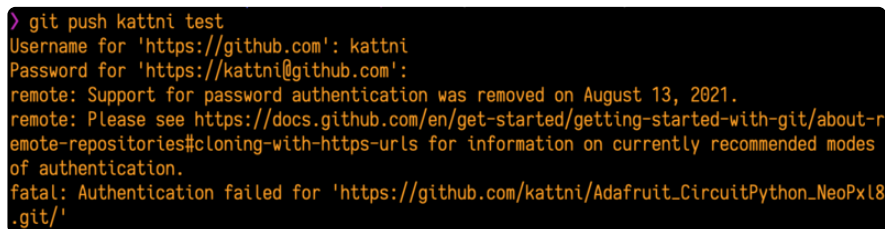
There are extra steps to pay attention to if you're working on the [CircuitPython core code \(https://adafru.it/tB7\)](https://adafru.it/tB7), versus contributing to a library. As we create new releases and begin to work on new versions of CircuitPython, the previous releases are moved to their own branches. In the event that you are adding something to one of the previous versions, the `push` command above may fail. Follow the instructions provided in the error message to properly `push` to your current working branch. For more details on `push`, please see the [git push documentation \(https://adafru.it/CXB\)](https://adafru.it/CXB).

GitHub Personal Access Token

You've used GitHub to fork a new repository. You've used Git to clone it to your computer, you've made your edits, and you're finally to the point where you are ready to push your changes back to GitHub. Remember, you needed to login to GitHub to access your account. The same goes for Git; it can't simply make changes to a repo on your GitHub account without first authenticating. That would be seriously insecure! This page explains how to use a personal access token to authenticate Git to interact with GitHub from your command line.

A **personal access token (PAT)** is an alternative to using a password for authentication. You should treat your PAT exactly as you would a password! They provide the same access in a more limited scope. GitHub opted to move to using PATs for authentication via command line, instead of allowing the continued use of password authentication. That means to be able to push your changes to your newly forked repository, you'll need to generate your own PAT.

Remember, you're here because your `push` failed to authenticate.



```
> git push kattni test
Username for 'https://github.com': kattni
Password for 'https://kattni@github.com':
remote: Support for password authentication was removed on August 13, 2021.
remote: Please see https://docs.github.com/en/get-started/getting-started-with-git/about-remote-repositories#cloning-with-https-urls for information on currently recommended modes of authentication.
fatal: Authentication failed for 'https://github.com:kattni/Adafruit_CircuitPython_NeoPx18.git/'
```

When you're done with the steps on this page, you'll run the same command again, with different results.

Let's get started!

A Note about GitHub Personal Access Token Types

There are currently two types of personal access tokens available within GitHub: fine-grained and classic. GitHub suggests using fine-grained whenever possible. As you'll see, this page explains classic token generation. Here's why.

There are a few differences between fine-grained and classic tokens that make fine-grained tokens more secure. One of those differences is that fine-grained token access is limited to specific repositories. This guide is intended to provide you with a work-flow that will apply to any public repository that you decide to contribute to. Using a fine-grained token would mean updating the token every time you decide to contribute to a new repository. Using a classic token means it will apply across the board to any repository you choose to fork.

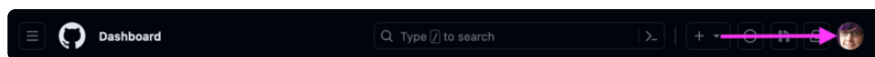
Further, and most relevant to this guide, GitHub has made it abundantly clear that fine-grained PATs are currently in beta, and therefore subject to change. Covering fine-grained token creation here could result in incorrect info at any time as GitHub continues to improve this feature.

Therefore, the choice was made to demonstrate creating a classic personal access token. It seems likely that fine-grained PATs will eventually replace classic, and at that time, this page will be updated, and this section removed.

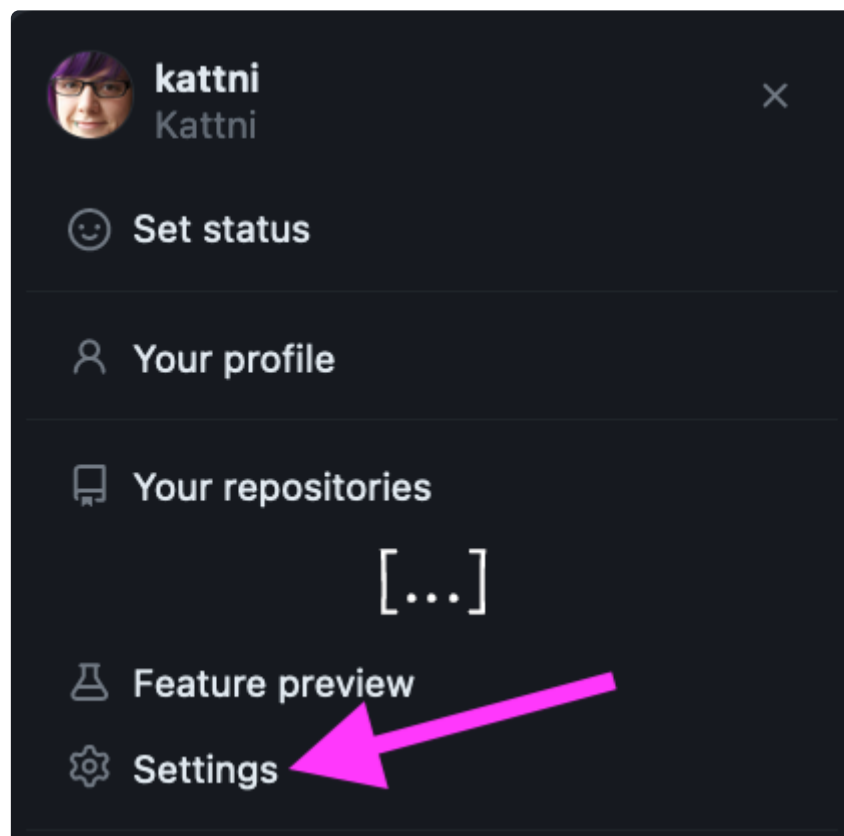
The Personal Access Tokens Menu

The first step is to navigate to the menu where you can begin generating your new token.

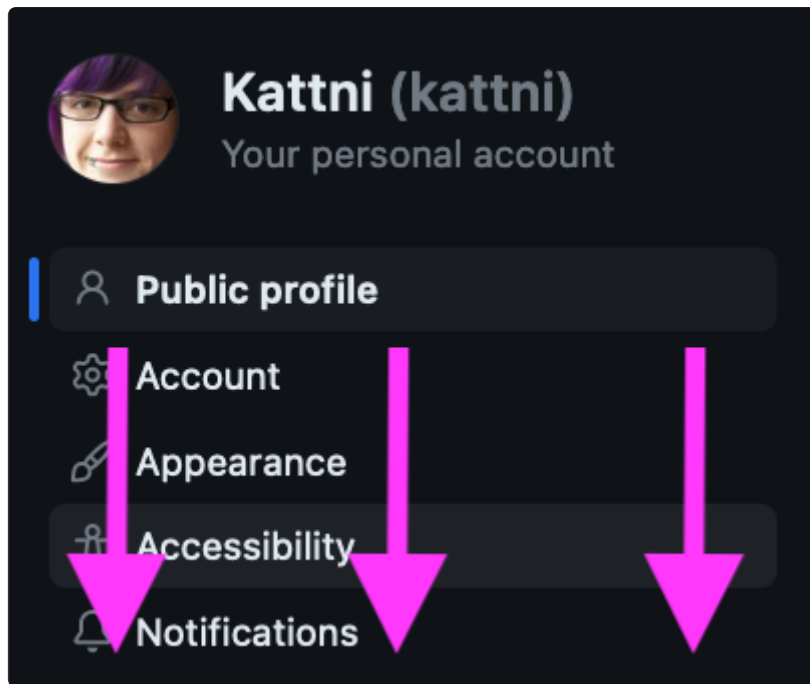
Click on your **profile image** in the upper right corner of any page on GitHub.



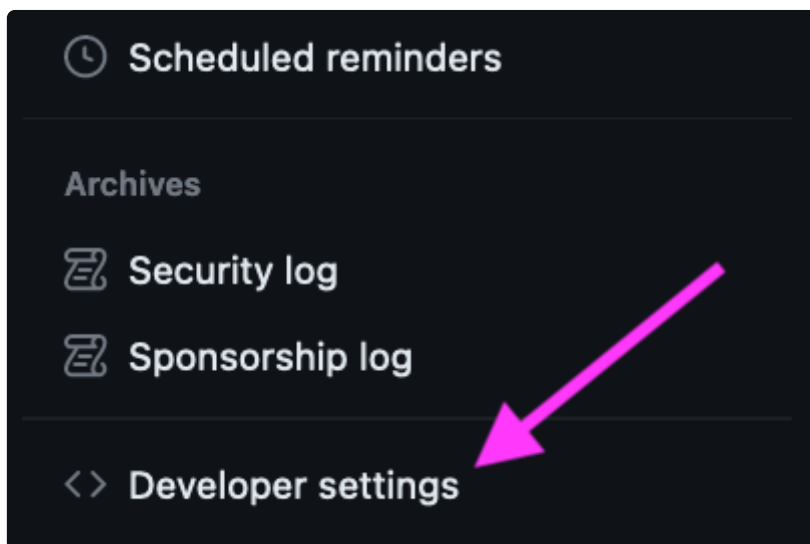
In the profile image menu, scroll to the bottom and click **Settings**.



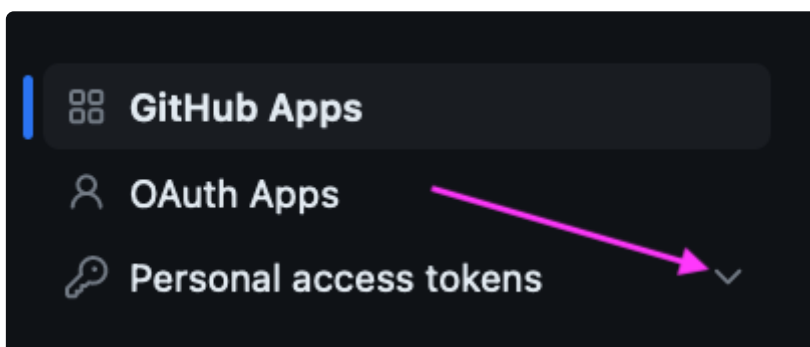
Settings has a lot of options! It's every setting, configuration, setup, etc for your entire account. Everything is listed down the left side. The option you're looking for is at the bottom. Scroll down to to the end of the list.



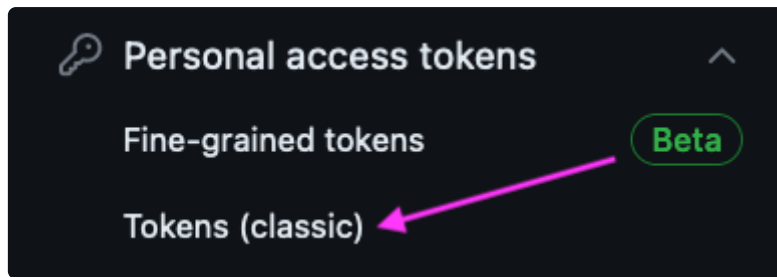
Click the **Developer Settings** option at the end of the list.



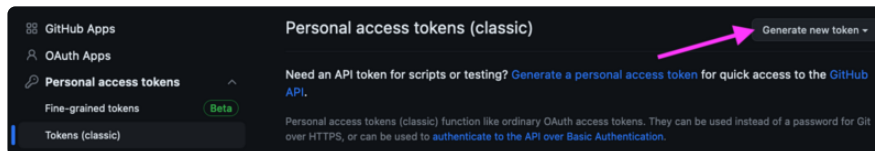
Developer Settings is a much shorter menu. Click the **arrow next to Personal access tokens** to expand the menu.



Choose **Tokens (classic)**.



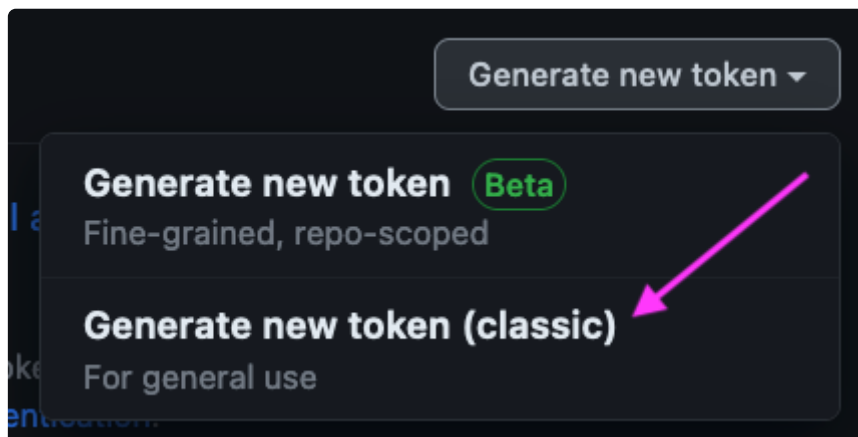
This opens the classic PAT list. If you have not yet created a classic PAT, you'll see the following. Click **Generate new token**.



If you have previously created a classic PAT, you'll have a "Revoke all" button as well as the Generate new token button. Click Generate new token.



From the Generate new token menu, choose **Generate new token (classic)**.

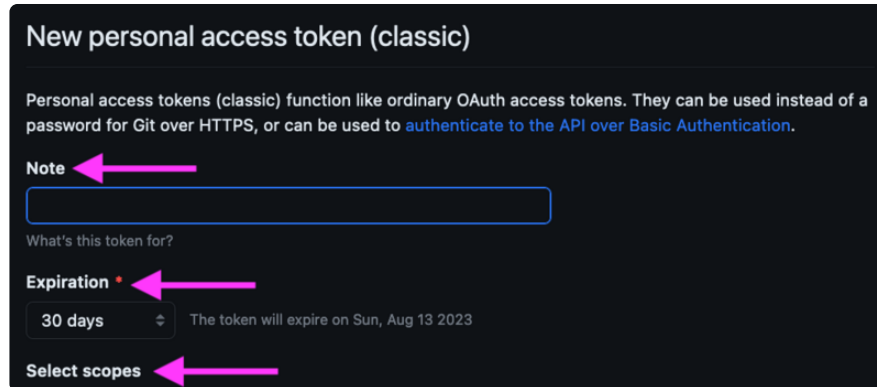


Now it's time to create your new token!

Generating a New GitHub Token (classic)

After clicking Generate new token (classic), you're taken to the **New personal access token (classic)** page. There are three sections you'll want to update: **Note**, **Expiration**, **Select scopes**.

While the only required section is Expiration (denoted by the red asterisk), for this purpose you will absolutely need to update Select scopes as well. Adding a note is strictly for you, and has no functional affect on the token, but your future-self will thank your current-self when you go back to view your token list, and it's clear which tokens are for what purpose.



The screenshot shows the 'New personal access token (classic)' form. It includes a text area for a 'Note', a dropdown for 'Expiration' (set to '30 days'), and a 'Select scopes' section. Three pink arrows point to the 'Note', 'Expiration', and 'Select scopes' labels.

New personal access token (classic)

Personal access tokens (classic) function like ordinary OAuth access tokens. They can be used instead of a password for Git over HTTPS, or can be used to [authenticate to the API over Basic Authentication](#).

Note

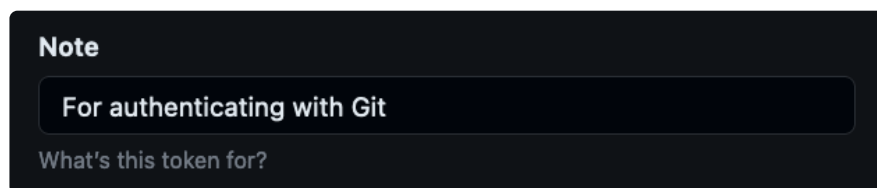
What's this token for?

Expiration *

30 days The token will expire on Sun, Aug 13 2023

Select scopes

For the **Note**, you'll want to add a phrase that makes it clear to you that you'll be using this token for authenticating Git with GitHub interactions. The following is a suggestion, but you can enter whatever works for you.



The screenshot shows the 'Note' field with the text 'For authenticating with Git' entered. Below the text is the label 'What's this token for?'.

Note

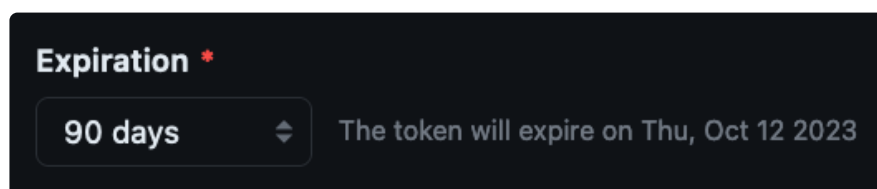
For authenticating with Git

What's this token for?

Choose an **Expiration**. This is the date upon which this token will expire, and if set, will require you to generate a new token at that point, and use it for its duration.

You can choose "No expiration", but you will receive a warning from GitHub informing you that it is less secure (spoiler: it is). No expiration means the token will continue to work until you manually revoke it. If it is ever compromised and you are unaware, this means that it will be exploitable until you decide arbitrarily to revoke it. Setting an expiration means that it will stop working every N-days, limiting its exploitability.

A good compromise is to choose 90 days. You will only have to regenerate a token four times per year, and you'll still be more secure.



The screenshot shows the 'Expiration' dropdown menu set to '90 days'. The text 'The token will expire on Thu, Oct 12 2023' is displayed next to the dropdown.

Expiration *

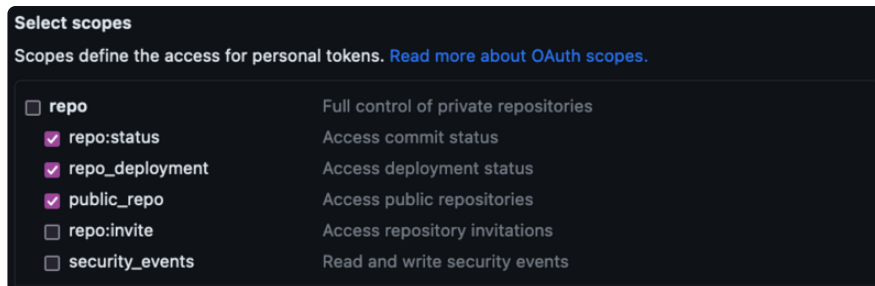
90 days The token will expire on Thu, Oct 12 2023

The final section is **Select scopes**. Scopes are designed to enable you to allow the token access to only the features you plan to use it for. This is also for security

reasons; if you gave every token full access, and it ends up compromised, the potential damage is much higher than if the scope is limited.

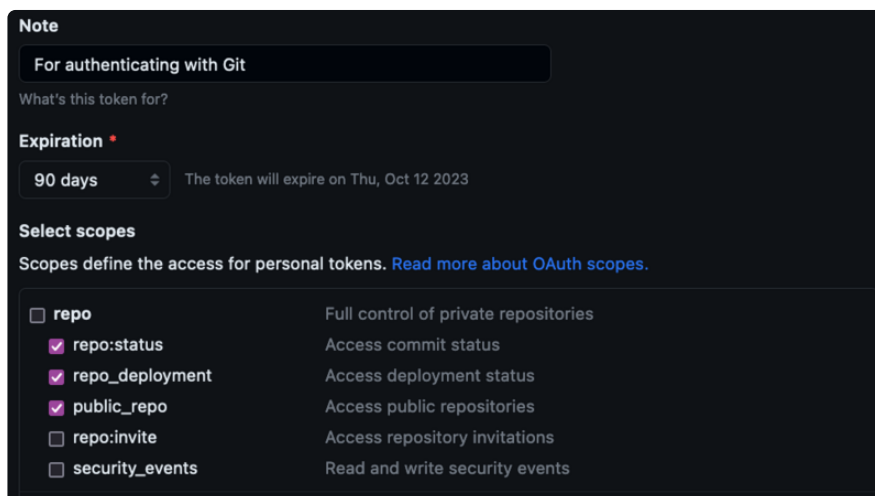
To be able to successfully interact with GitHub using Git via command line, you need only three options from the entire list. Check the following boxes nested under **repos**:

- **repo:status**
- **repo_deployment**
- **public_repo**



Select scopes	
Scopes define the access for personal tokens. Read more about OAuth scopes.	
<input type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events

Now, verify that all of the information you provided is correct. Your **New personal access token (classic)** page should now resemble the following.



Note

For authenticating with Git

What's this token for?

Expiration *

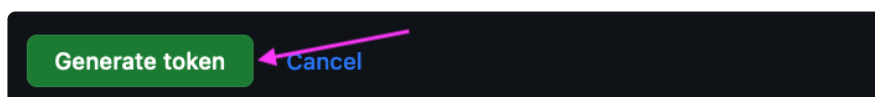
90 days The token will expire on Thu, Oct 12 2023

Select scopes

Scopes define the access for personal tokens. [Read more about OAuth scopes.](#)

<input type="checkbox"/> repo	Full control of private repositories
<input checked="" type="checkbox"/> repo:status	Access commit status
<input checked="" type="checkbox"/> repo_deployment	Access deployment status
<input checked="" type="checkbox"/> public_repo	Access public repositories
<input type="checkbox"/> repo:invite	Access repository invitations
<input type="checkbox"/> security_events	Read and write security events

When you're happy with it, scroll down to the bottom of the page, and click **Generate token**.



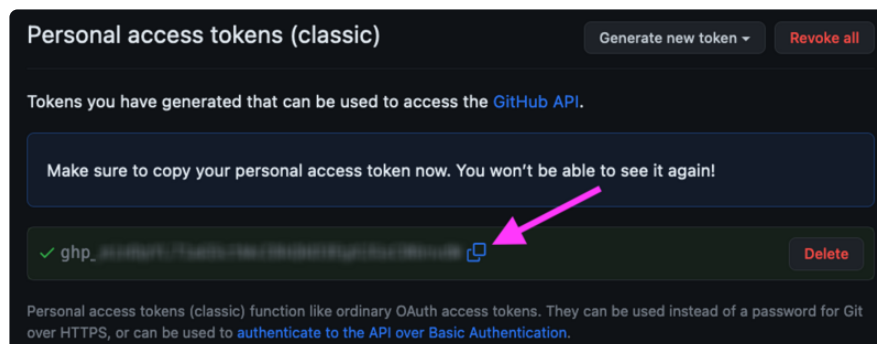
Generate token Cancel

Your New Token

You'll be returned to the **Personal access tokens (classic)** page. This time, however, you'll see your new token.

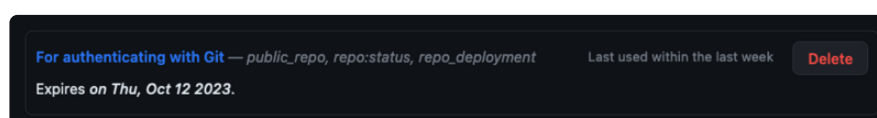
You must **immediately** copy your new token. **This is the only time the token itself will be displayed.** Copy it by clicking on the copy button (the two-box icon to the right of the token). Store it someplace secure to which you have access, such as a password manager. You will need to use it later to authenticate Git.

You must copy your new token NOW. This is the ONLY time the token itself will be displayed! If you do not copy it now, or you lose it, you will need to go through these steps again to generate another new token.



Once you navigate away from that page, or refresh it, you will no longer have access to the token.

When you return to this page, you will see only the information about your token. On the left side, you'll find the Note text as a link, followed by the scopes you selected. Below the Note text is the date that the token will expire. On the right side you'll see when you last used the token, and a Delete button.



There is no way to view the token again. You may delete it if you no longer need it.

Authenticating via Command Line

It's time to go back to your terminal program. Run the exact same **push** command you ran earlier. When asked for your username, enter it as before. This time, however, **when asked for your password, enter your new personal access token.**

```
> git push kattni test
Username for 'https://github.com': kattni
Password for 'https://kattni@github.com':
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
remote:
remote: Create a pull request for 'test' on GitHub by visiting:
remote:   https://github.com/kattni/Adafruit_CircuitPython_NeoPx18/pull/new/test
remote:
To https://github.com/kattni/Adafruit_CircuitPython_NeoPx18.git
 * [new branch]      test -> test
```

Success! Now you can continue to the next step, creating your pull request!

Further Information

GitHub has excellent documentation on [Managing your personal access tokens](https://adafru.it/19dY) (<https://adafru.it/19dY>). It includes the process to store your PAT locally so you don't have to reenter it when you're working with Git. If you're interested in more information about how to maintain and manage your new PAT, the GitHub docs have you covered!

Add Author and License Information

Adafruit is standardizing on adding author and license information to all code files. For Arduino, this would be all .ino, .h, and .cpp files. For CircuitPython, it would be all .py files.

The information goes at the top of each code file before any other comments or code.

Please put the current year and the author(s) first and last name in the first line. In the third line, generally Adafruit code is MIT licensed. If you code derives from code under CC or other licenses, please list them, although **MIT is the preferred license**.

The phrase "for Adafruit Industries" is for Adafruit authors and is not required for general authors.

Arduino

```
// SPDX-FileCopyrightText: YYYY Your Name for Adafruit Industries
//
// SPDX-License-Identifier: MIT
```

An example:

```
// SPDX-FileCopyrightText: 2021 Anne Barela for Adafruit Industries
//
// SPDX-License-Identifier: MIT
```

First line of code or additional comments

or

```
// SPDX-FileCopyrightText: 2022 Jane Doe  
//  
// SPDX-License-Identifier: MIT
```

If there is more than one author, you can have multiple lines for SPDX-FileCopyrightText, one below the other each listing the date and author as appropriate:

```
// SPDX-FileCopyrightText: 2017 Limor Fried/ladyada for Adafruit  
Industries  
// SPDX-FileCopyrightText: 2017 Phillip Burgess for Adafruit  
Industries  
//  
// SPDX-License-Identifier: MIT
```

CircuitPython

CircuitPython uses the Python comment `#` instead of the C comment `//`. Also the latest versions of the Lint/CI checker want a description delimited by three single quotes beginning and end:

```
# SPDX-FileCopyrightText: YYYY Your Name for Adafruit Industries  
#  
# SPDX-License-Identifier: MIT  
'''MyExample is a program to play MP3 sound files'''
```

An example:

```
# SPDX-FileCopyrightText: 2021 Anne Barela for Adafruit Industries  
#  
# SPDX-License-Identifier: MIT  
'''MyExample is a program to draw pictures'''
```

First line of code or additional comments

SPDX

Documenting information in this fashion is consistent with The Software Package Data Exchange® (SPDX®). SPDX is an open standard for communicating software bill of material information, including components, licenses, copyrights, and security references. SPDX reduces redundant work by providing a common format for companies and communities to share important data, thereby streamlining and improving compliance.

The SPDX specification is an international open standard [ISO/IEC 5962:2021 \(https://adafru.it/VrB\)](https://adafru.it/VrB).

The list of valid license types are [on the SPDX website here \(https://adafru.it/VrC\)](https://adafru.it/VrC).

Please keep to the formats listed above unless instructed by Adafruit staff.

Troubleshooting

- Be sure the comments are left justified and worded close to the example text including spaces.
- Be sure you used the right comment mark for the language you are coding with.
- Check that the license given is valid in the list [here \(https://adafru.it/VrC\)](https://adafru.it/VrC)
- If you are not writing for Adafruit, the phrase "for Adafruit Industries" is not required.

Create Your Pull Request

You've committed your changes and pushed them to your fork. You're ready to submit your changes to the original project for review. This means you're ready to put in a **pull request**.

A **pull request**, or **PR**, is exactly that: a **request** to **pull** your changes into the original project code. Basically, you're asking the owner of the original project to include your new changes. When changes are included in a project, it's called a **merge**.

Completing a pull request involves **merging** the pull request into the original project.

A pull request isn't a single step, however. It's a process. You'll create your PR, submit any fixes necessary for the checks to pass, wait for review, submit any or discuss changes requested in the review, and then wait for your code to be merged into the project. Sometimes the review discussion might be quite involved. Not all PRs will be accepted. This is why it's important to submit a PR earlier rather than later so you can get feedback earlier on in the development process.

This section of the guide will cover creating your pull request. Let's get started!

Creating a Pull Request from Your **push**

Let's take one more look at the results of that last **git push**. Take a look at the highlighted section.

```

> git push kattni f-strings-and-params
Enumerating objects: 8, done.
Counting objects: 100% (8/8), done.
Delta compression using up to 12 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (6/6), 752 bytes | 752.00 KiB/s, done.
Total 6 (delta 4), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (4/4), completed with 2 local objects.
remote:
remote: Create a pull request for 'f-strings-and-params' on GitHub by visiting:
remote:   https://github.com/kattni/Adafruit_CircuitPython_Dash_Display/pull/new/f-strings-and-params
remote:
To github.com:kattni/Adafruit_CircuitPython_Dash_Display.git
 * [new branch]      f-strings-and-params -> f-strings-and-params

```

The highlighted section contains the following text, except with the applicable GitHub user ID, repository name, and branch name.

```

Create a pull request for 'your-branch-name' on GitHub by visiting:
https://github.com/youruserID/Repo_Name/pull/new/your-branch-name

```

That's convenient! There are instructions and a URL are right in the **push** results. You can simply click on the link, or copy and paste it into your browser to get started.

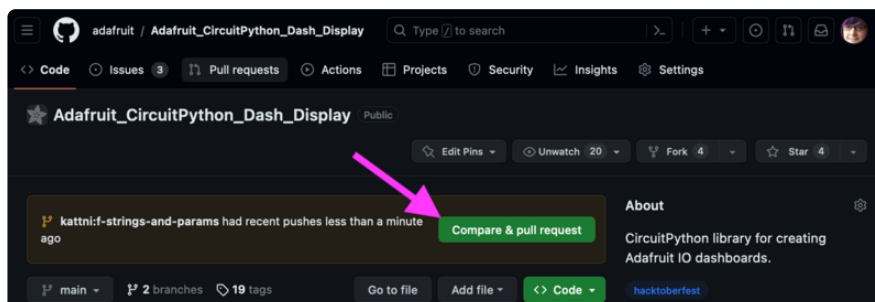
There are plenty of situations where you will not create a pull request immediately after your first **push** to your fork. In those cases, you might not have that direct URL by the time you're ready to open your PR. Not a problem! You can also open a PR through GitHub.

Creating a Pull Request through GitHub

There are a couple of ways to create a pull request through GitHub, depending on how recently you pushed your code. If you pushed recently, you can create it from the original repository. If it's a bit, you'll want to create it from your fork.

Creating a PR from the Original Repository

Open your browser and navigate to the original repository, in this case, the Adafruit version of the library that you're contributing to. You'll see a box along the top with your username followed by your branch name, indicating that the branch had recent pushes. To create the PR, click the **Compare & pull request** button.

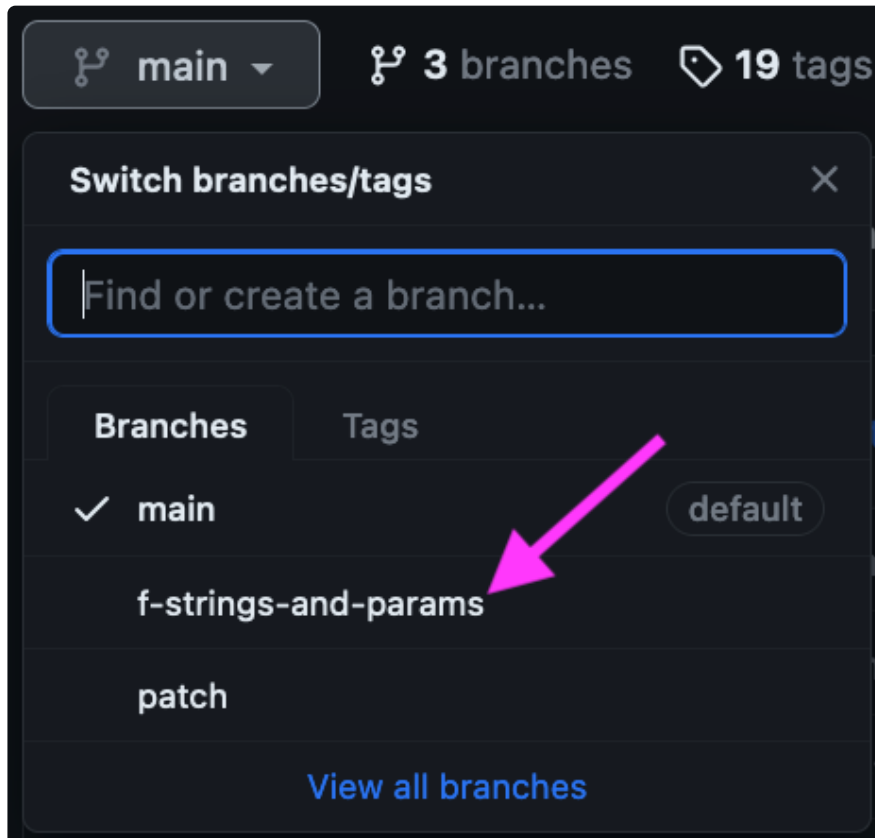


Creating a PR from Your Fork

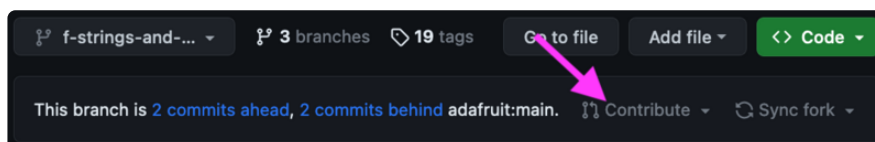
Open your browser and navigate to your forked copy of the library you're contributing to. Above the file list, you'll see a dropdown that shows the main branch. Click on **main**.



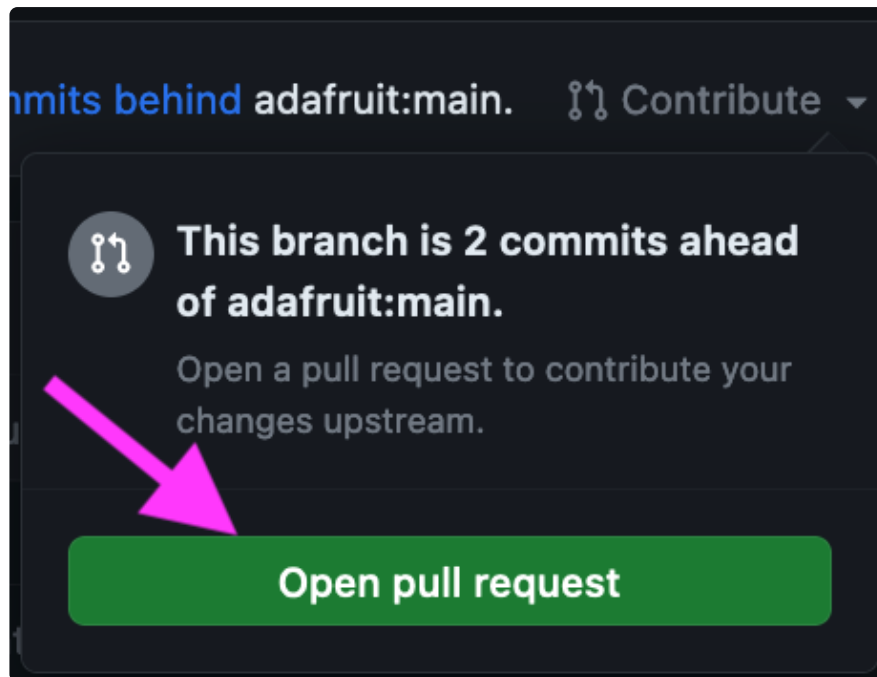
From the resulting list, choose your branch name. Mine is **f-strings-and-params**.



The branch will now show your branch name, and you'll see a new section below it. Click **Contribute**.

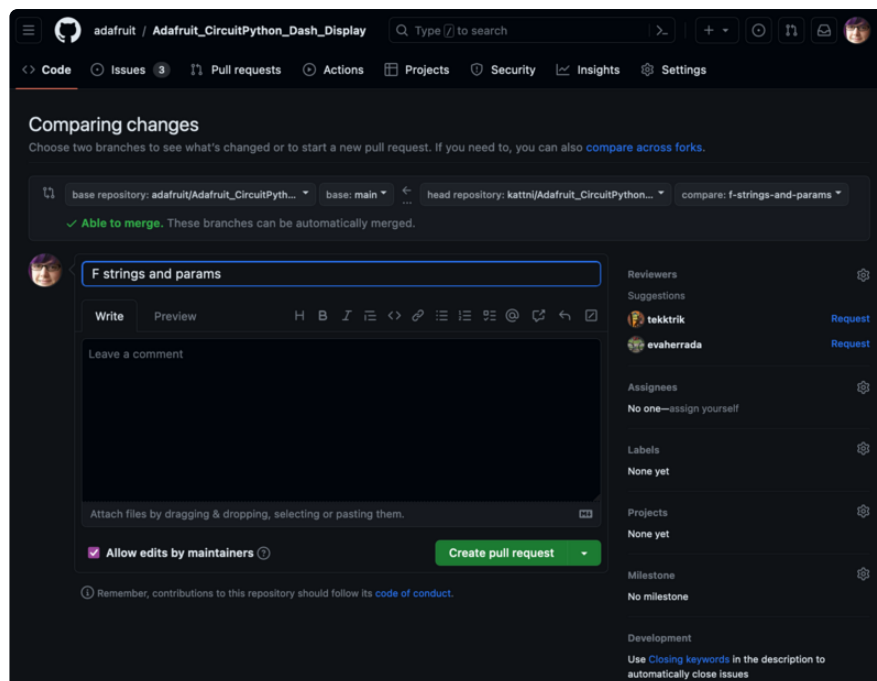


Click **Open pull request**.



Open a Pull Request

The next thing to do is open the pull request. The initial page will look something like this.

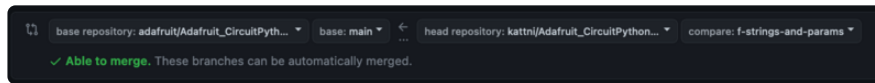


Let's break it down!

Automatic Merge Verification

The first section will let you know whether your request is able to be automatically merged. If it's not, that means someone else already made changes to the same section of code that you did, and you'll need to update your code to match the

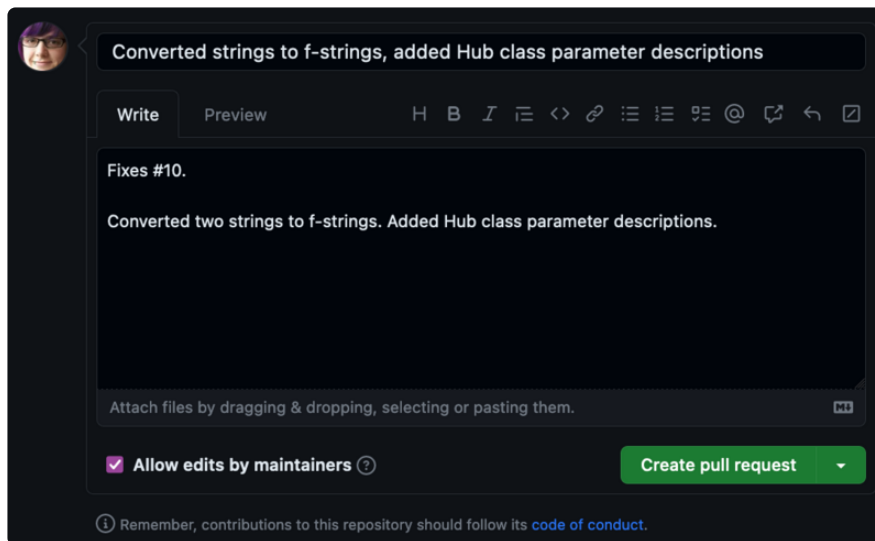
already existing changes before you can submit the pull request. It's possible to submit a PR that isn't able to be automatically merged, but often the owner of the project will ask you to update your code first anyway. So it's good practice to not submit until that section says **Able to merge**.



Pull Request Title and Message

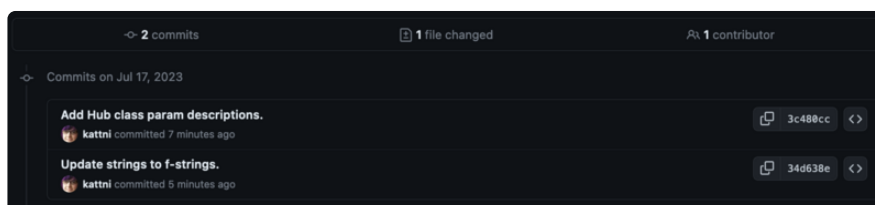
The next section is where you'll enter your pull request message. Ideally you'll title it something that quickly describes what you changed. Then you can include more details in the message body. If you made a single commit, they may already be populated by your commit message. If you made multiple commits, it may simply be populated by your branch name. You can change them regardless if you'd like to be more descriptive.

Since I made multiple commits, it included only my branch name. So I've chosen to update it to include much more detailed information about my pull request.



Commit List

The next section includes your list of commits. The top details the number of commits, how many files were changed, and the number of contributors. I made two commits, changed one file, and am the sole contributor.



Your Changes

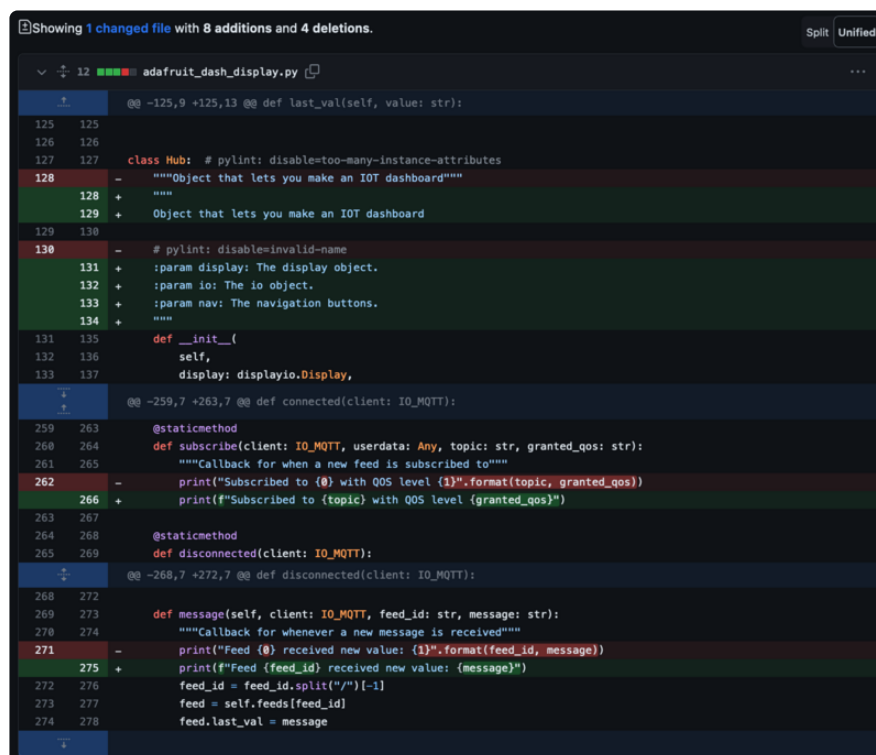
The last section shows you your changes. Like `git diff`, it will show you only the code surrounding your changes.

Above the changes are the number of files shown, and the total number of additions and deletions included in this change.

Then you'll see the name of the file you changed. The code you've added will show up in green. Any code you deleted will show up in red. Be aware, if you change an entire code block, it will show the original code in red, and your new code in green, even if you didn't remove the code contained within the block. This doesn't mean the code you changed was deleted! It's simply how the changes are shown here.

You can see here that I added the parameter descriptions, and updated the strings.

There are two ways to view changes on GitHub: **unified view** and **split view**. GitHub defaults to unified view, which is shown below. It shows the original and updated code in one column, very similar to `git diff`, which is very familiar to many folks. However, it can be a struggle for new folks to follow. That's where split view can help.

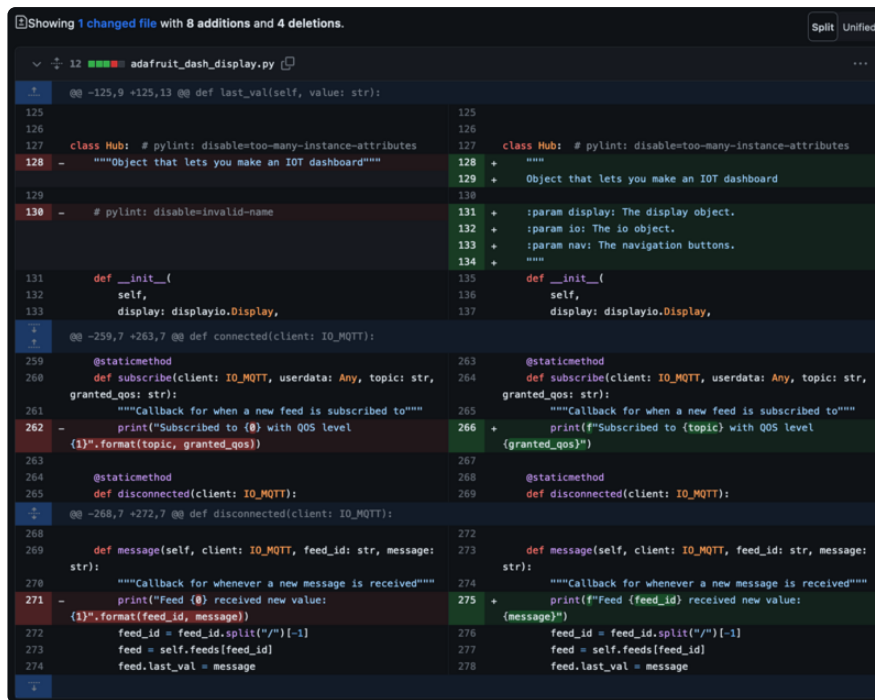


```
Showing 1 changed file with 8 additions and 4 deletions. Split Unified
adafuit_dash_display.py
@@ -125,9 +125,13 @@ def last_val(self, value: str):
125 125
126 126
127 127 class Hub: # pylint: disable=too-many-instance-attributes
128 - """Object that lets you make an IOT dashboard"""
128 + """
129 + Object that lets you make an IOT dashboard
129 130
130 - # pylint: disable=invalid-name
131 + :param display: The display object.
132 + :param io: The io object.
133 + :param nav: The navigation buttons.
134 + """
131 135 def __init__(
132 136     self,
133 137     display: displayio.Display,
@@ -259,7 +263,7 @@ def connected(client: IO_MQTT):
259 263 @staticmethod
260 264 def subscribe(client: IO_MQTT, userdata: Any, topic: str, granted_qos: str):
261 265     """Callback for when a new feed is subscribed to"""
262 - print(f"Subscribed to {topic} with QOS level {1}.format(topic, granted_qos)")
266 + print(f"Subscribed to {topic} with QOS level {granted_qos}")
263 267
264 268 @staticmethod
265 269 def disconnected(client: IO_MQTT):
@@ -268,7 +272,7 @@ def disconnected(client: IO_MQTT):
268 272
269 273 def message(self, client: IO_MQTT, feed_id: str, message: str):
270 274     """Callback for whenever a new message is received"""
271 - print(f"Feed {0} received new value: {1}.format(feed_id, message)")
275 + print(f"Feed {feed_id} received new value: {message}")
272 276     feed_id = feed_id.split("/")[-1]
273 277     feed = self.feeds[feed_id]
274 278     feed.last_val = message
```

To switch to split view, click **Split**, found on the right side above the changes.



Split view shows you the original code in a column on the left, and the updated code in a column on the right. This may be easier for you to follow.

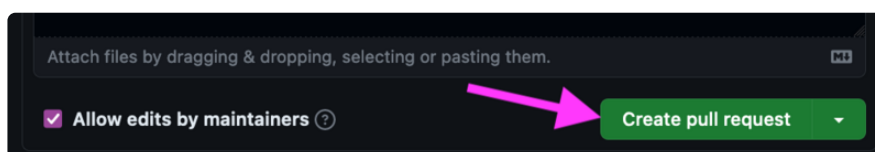


Once you pick a view, GitHub will remember your choice, and the next time you see changes, they'll be in whichever format you prefer.

Create Your Pull Request

Go through each of these sections and make sure they're all correct. Did you include all the commits you meant to? Do the changes show all of the changes you intended to make? Did you find a mistake? If you find anything you missed or need to change, back out of the pull request and finish up what you need to. Then start the process again.

If you're happy with everything you see, you're ready to open your pull request. Under where you entered your description, click the **Create pull request** button.



You've created your pull request! The next section will cover what happens during the open pull request. Let's take a look!

Your Open Pull Request

Now you've created your pull request and you're ready to continue the process. This section shows you how to ensure your PR is ready for review. It details the important features of your new PR. It covers how to read the logs generated by the check

system, and deal with the built in code checks failing. Finally, you'll learn how to submit code to an active PR.

An important part of software development is ensuring your code is consistent. CircuitPython, like any open source project you might contribute to, has expectations and standards, and if you want your contribution to be accepted, you'll need to work within them. Our responsibility in this is to clearly communicate our expectations, and provide you with the tools necessary to guarantee your code complies. To that end, the CircuitPython libraries all have a series of checks to verify that your code fits with our standards.

Adafruit uses a GitHub tool called **Actions**. For CircuitPython libraries, it is used to check a number of things including:

- Are there any syntax issues with the code?
- Is the code style as expected?
- Is the code formatted properly?
- Does the code include an appropriate license and attribution?
- Does the documentation build?

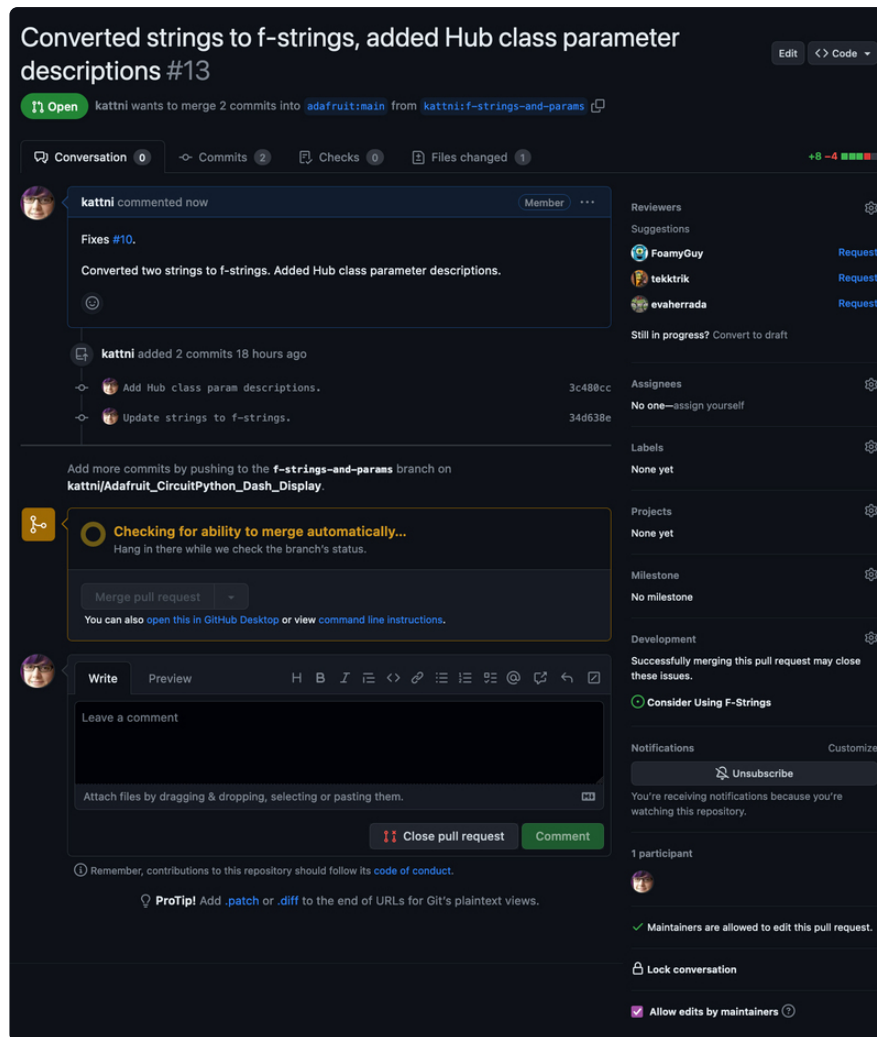
These things help us by automating a significant part of the review process by ensuring that the basic standards are being met. It means the folks reviewing can spend more time in the review process working directly with you on your contribution, instead of taking the time to verify these requirements.

The fact is, you're going to run into these checks failing, no matter how many times you go over your code before submitting. So, it's important to be able to read the log generated by Actions. You'll be able to understand what the failures mean, and use the log to more effectively resolve the issues in your contribution.

Your Pull Request Explored

Welcome to your pull request! Once you've clicked the **Create pull request** button, it will automatically take you to your newly created PR.

The main page is going to look something like the following. There's a lot going on here! By the time you're done with this section, you'll be familiar with all the important features of your PR. Let's take a look!

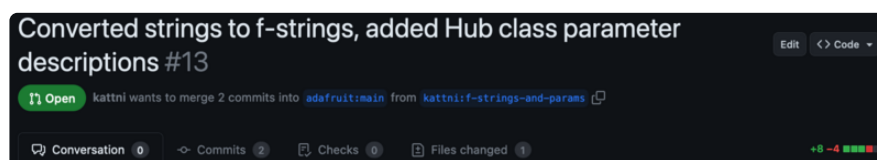


Header

The PR header begins with the PR title, and the number assigned to it when it was created. My PR is **#13**. Below that is a quick view of whether the PR is open, closed or merged; it is currently **Open**. Next to the status, is the current PR content summed up in a human-readable sentence.

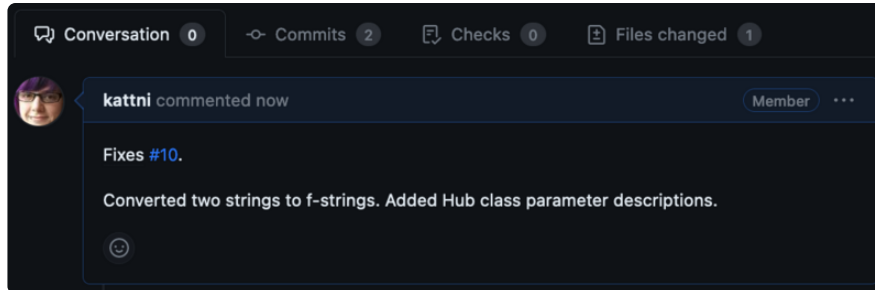
The bottom of the header has four tabs: **Conversation**, **Commits**, **Checks**, and **Files changed**. The tabs you'll spend the most time in are Conversation and Files changed.

At the bottom right of the header are some red and green numbers and blocks. The green shows you the number of additions to the code in your contribution, and the red shows you the number of deletions. If you add more changes to your PR, these numbers will update to match the overall statistics. They're not particularly important, but they are fun!



Comment

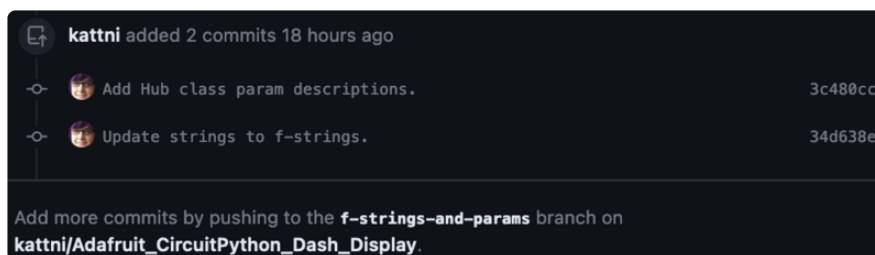
The first thing in the Conversation tab is your **PR comment**. You'll see your profile picture, your name, and when the comment was added. I clicked create a moment ago, so it's showing that I "commented now".



Commit List

Below your comment is the **list of commits currently included in this PR**. You'll see who added the commits and when. Each entry in the list begins with the profile picture of the commit author, followed by the commit title, and on the far right, the last seven characters of the commit hash.

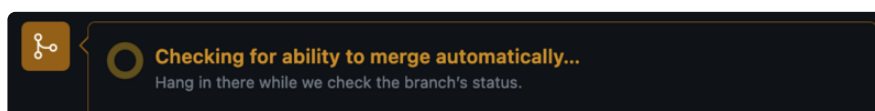
As you can see, I made the changes quite a while before I created this PR.



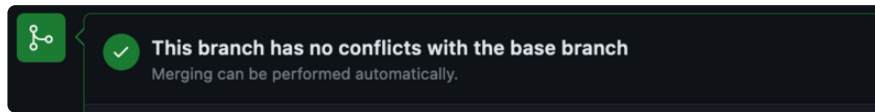
Status and Merge Info

Below the commit list is the **PR status**. This section reflects the current status of the PR at any given time. This is different from the open status reflected in the header. This status tells you what condition the PR is in, whether checks are running or changes have been requested. You can always check here to see where things are at.

There are many statuses. If you scroll down as soon as you create the PR, you'll see the initial status, which is, "**Checking for ability to merge automatically...**" A PR can be merged automatically if there are no changes in the PR that conflict with the existing code.

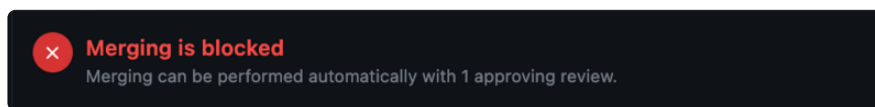


If there are no issues, the next status will be, "This branch has no conflicts with the base branch".



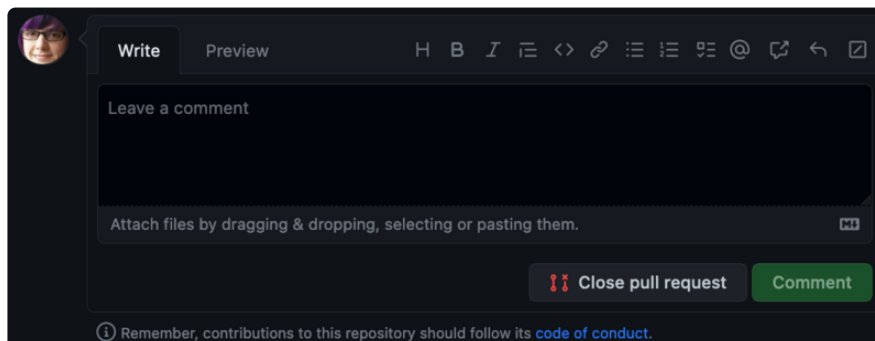
If there were issues, you would have been notified during PR creation, and given the opportunity to resolve them before creating your PR.

You may also see the following on certain repos that have merge restrictions enabled. These are usually highly utilised libraries where changes must be considered carefully, as breaking changes would cause serious, wide-spread issues.



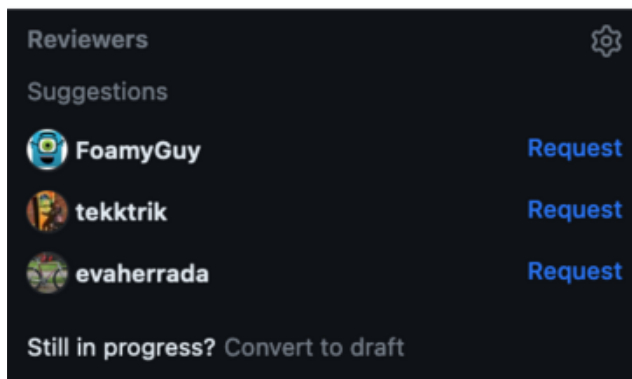
Leave a Comment

The last section of the main column is the comment dialogue box. This is where you would post another standalone comment to the PR, or an overall reply to a review comment. Here you can also close your PR if you desire, with or without a comment. Remember, it can always be reopened!

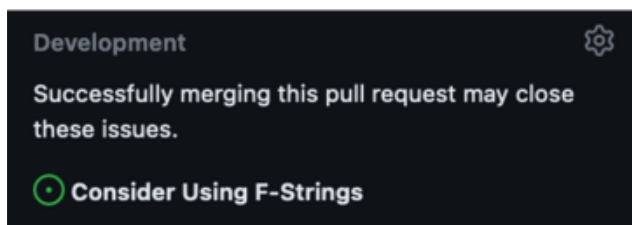


The Right Column

Here's a quick look at a couple of sections of the right column. The rest will be skipped as they are less relevant to this guide.



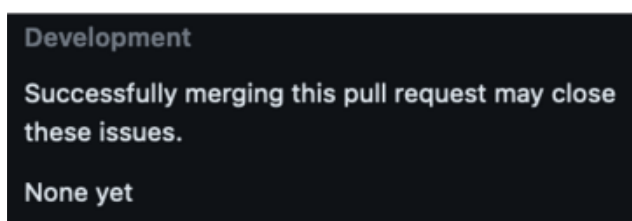
The top section is **Reviewers**. Here you will find the individuals or teams from whom a review has been requested, or those who are actively reviewing your PR. You will likely not be able to request a review on your own PR, but don't worry, we'll take care of it for you!



The sixth section is **Development**. Here, if applicable, you'll find a list of issues by name.

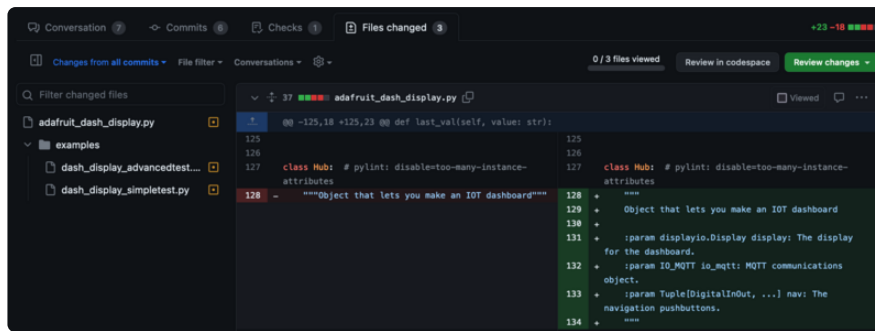
If you look back at my PR comment, you'll see the first thing I include is, "Fixes #10." Looking through that repository, you'd find that issue 10 refers to the f-strings. **As my PR addressed that issue, I can put specific language in my PR comment, and it will automatically close the issue when the PR is merged!**

If there are no associated issues, it will simply say "None yet".

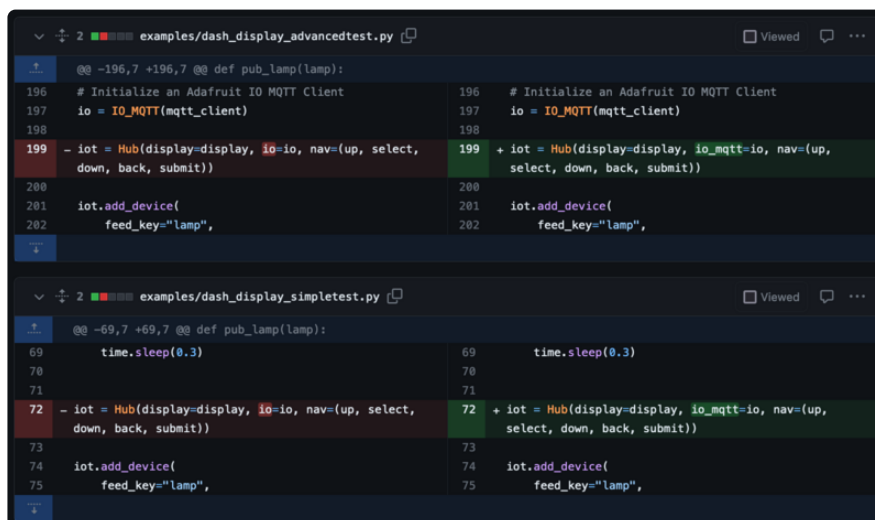


The Files Changed Tab

The **Files changed** tab shows you the same comparison you saw when you created your pull request. In the case of only one file, it will look very similar to the PR creation comparison. When you've edited more than one file, in addition to the change comparison, it will also include a new column on the left showing the file names and directory structure. Below, one file in `/`, and two files in `examples/` were edited.



To see the other two files, you would scroll down through the comparison until you find them.



That's your new pull request! Now you know what you're looking at, and will hopefully find it easier to make your way around as the process continues.

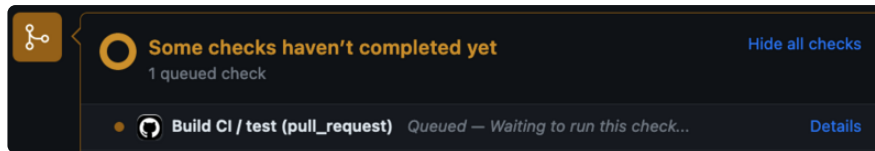
Time for Action!

Once your pull request is created, Actions automatically begins going through the steps to check your code. You can keep up with where things are at by checking the status section! Know that while the status may say there is only one check running, that check is verifying a large series things at once.

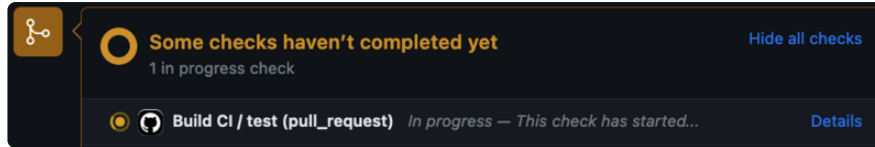
Some Checks Haven't Completed Yet

The checks can sometimes take a bit of time to run. During that time, the status will be, **"Some checks haven't completed yet"**.

You'll notice though, that now there's a new section in the status area. This section shows the check (or checks) that are being run on your pull request, as well as the current status of the check. Initially, it may show as queued.

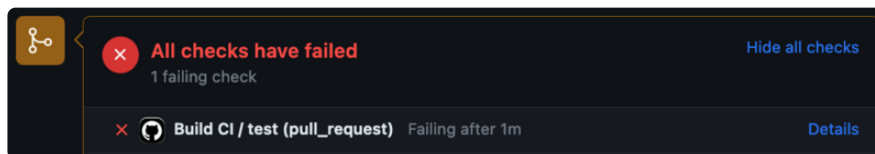


Once the check is running, the status will update.



All Checks Have Failed

The check has completed, with the status, **"All checks have failed"**. In the event of a failure, you'll need to work through it before the review process can begin.

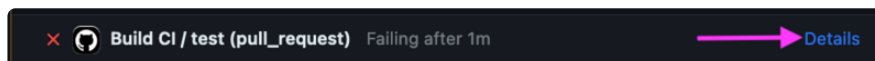


The most recent commit in the commit list shows a red X next to it, indicating the same.



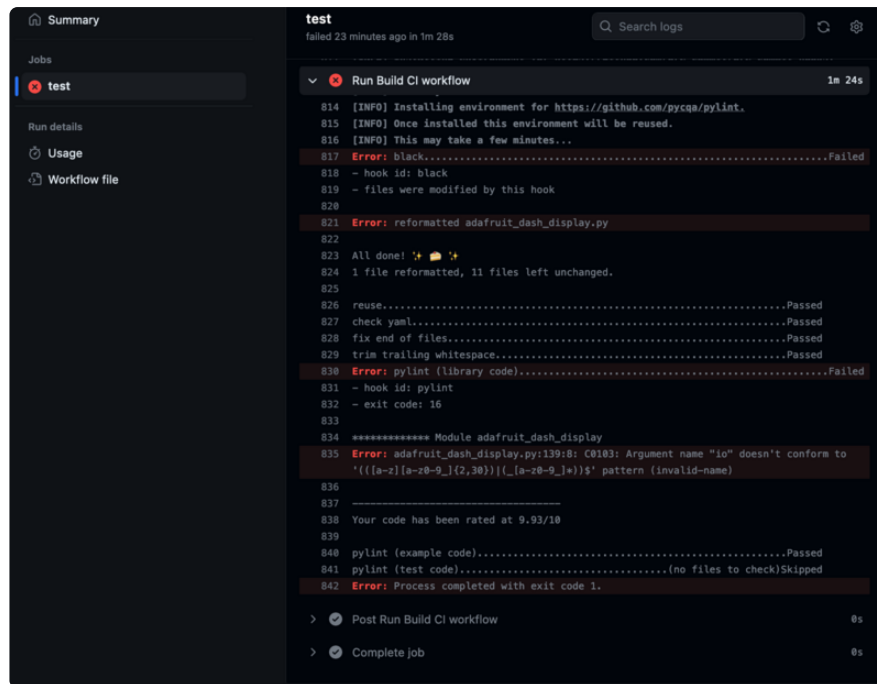
This is going to happen to you. Consider it to be a normal part of the process. We've all dealt with it. The more you go through it, the more you'll learn, and eventually it will happen less. But inevitably you'll miss something, and the Actions checks are here to let you know. Let's take a look at how to get that information from Actions.

Click on the **Details** link found in the section letting you know that the checks have failed.



This will take you to the Actions log. Lucky for us, it scrolls you directly to the place in the log where the failure occurred. Here's what you'd find if you clicked Details on my PR. There is a right column that you won't do much with. The info that you want is on the left. It looks really involved, but once you're through this section, you'll know exactly what you're looking for.

In my case, there are two code checks that failed. How can I tell? Let's take a closer look.



```
Summary
Jobs
  test
Run details
Usage
Workflow file

test
failed 23 minutes ago in 1m 28s
Search logs

Run Build CI workflow 1m 24s
814 [INFO] Installing environment for https://github.com/pycqa/pylint.
815 [INFO] Once installed this environment will be reused.
816 [INFO] This may take a few minutes...
817 Error: black.....Failed
818 - hook id: black
819 - files were modified by this hook
820
821 Error: reformatted adafruit_dash_display.py
822
823 All done! 🎉 🍌 🎉
824 1 file reformatted, 11 files left unchanged.
825
826 reuse.....Passed
827 check yaml.....Passed
828 fix end of files.....Passed
829 trim trailing whitespace.....Passed
830 Error: pylint (library code).....Failed
831 - hook id: pylint
832 - exit code: 16
833
834 ***** Module adafruit_dash_display
835 Error: adafruit_dash_display.py:139:8: C0103: Argument name "io" doesn't conform to
      '([a-z][a-z0-9]{2,30})|([a-z0-9]{1})$' pattern (invalid-name)
836
837 -----
838 Your code has been rated at 9.93/10
839
840 pylint (example code).....Passed
841 pylint (test code).....(no files to check)Skipped
842 Error: Process completed with exit code 1.

> Post Run Build CI workflow 0s
> Complete job 0s
```

I'm going to look at the errors. The first indicates that the **black** check has failed. It also tells me which file Black failed on.

[Black \(https://adafru.it/MF4\)](https://adafru.it/MF4) is our code formatter; it ensures that all library code is formatted consistently. When Black fails, it means there is a formatting issue in my code that I will need to resolve.

```
817 Error: black.....Failed
818 - hook id: black
819 - files were modified by this hook
820
821 Error: reformatted adafruit_dash_display.py
822
823 All done! 🎉 🍌 🎉
824 1 file reformatted, 11 files left unchanged.
```

You'll see the next four checks passed. Excellent!

However, the **pylint (library code)** check also failed. The log provides the [Pylint error \(https://adafru.it/19dZ\)](https://adafru.it/19dZ), which includes all the information you need to fix it up.

[Pylint \(https://adafru.it/FDc\)](https://adafru.it/FDc) is our linter, which is used to check your code for syntax and style. When Pylint fails, it means there is a stylistic issue in my code that I will need to resolve.

```
830 Error: pylint (library code).....Failed
831 - hook id: pylint
832 - exit code: 16
833
834 ***** Module adafruit_dash_display
835 Error: adafruit_dash_display.py:139:8: C0103: Argument name "io" doesn't conform to
      '([a-z][a-z0-9]{2,30})|([a-z0-9]{1})$' pattern (invalid-name)
836
837 -----
838 Your code has been rated at 9.93/10
```

Now that I know what caused the PR checks to failed, I can work through it. [This guide \(https://adafru.it/19e0\)](https://adafru.it/19e0) covers the process of handling Pylint and Black errors.

We highly suggest installing pre-commit which automatically runs these checks before you even commit, which means avoiding them in your pull request. I deliberately avoided pre-commit for this PR to show how to read the error logs.

Installing pre-commit in the CircuitPython library directory on your computer will help you avoid dealing with errors on your pull request.

Pushing the Fix to Your PR

You found the errors, and you fixed them on your computer. Now, you need to submit them. **You don't have to make another pull request to do this!** Simply follow the same steps you followed previously, push to the same branch, and your new commit will appear! **Pushing a commit to the current branch will update the open PR.** Let's make it happen.

Once you're satisfied that you've made the necessary fixes, you're going to `status`, `add`, `commit`, and `push`.

```
kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params *! (3
> git status
On branch f-strings-and-params
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   adafruit_dash_display.py

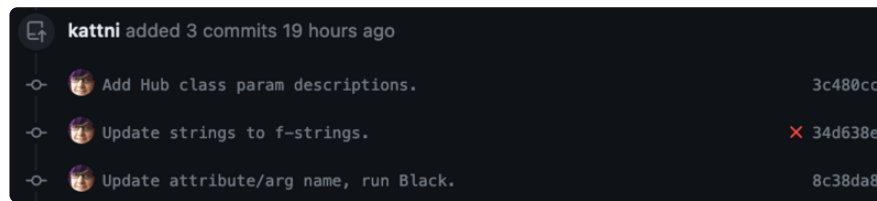
no changes added to commit (use "git add" and/or "git commit -a")

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params *! (3
> git add .

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params ** (3
> git commit -m "Update attribute/arg name, run Black."
[f-strings-and-params 8c38da8] Update attribute/arg name, run Black.
 1 file changed, 14 insertions(+), 13 deletions(-)

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params (8c38
> git push kattni f-strings-and-params
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 393 bytes | 393.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_Dash_Display.git
 34d638e..8c38da8 f-strings-and-params -> f-strings-and-params
```

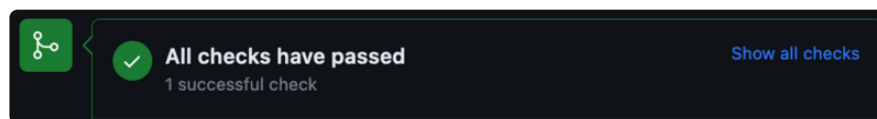
Now, you can go back to your pull request and check the commit list. There is the new commit!



The status will update to indicate the checks are running. **Every time you add a new commit to a pull request, Actions will automatically run the checks.** The checks can fail at any time during the PR process when multiple commits are involved. Automatically checking each commit ensures that, even if you get the checks passing with a particular commit, you don't accidentally introduce new issues with a later one. Let's see what happens!

All Checks Have Passed

There you have it! The PR status is, "**All checks have passed**", which means your PR is ready to go. You'll see a green check next to the commit hash in the commit list as well.



Your pull request is now ready for review! The next section covers the process of receiving a review.

Receiving a Review

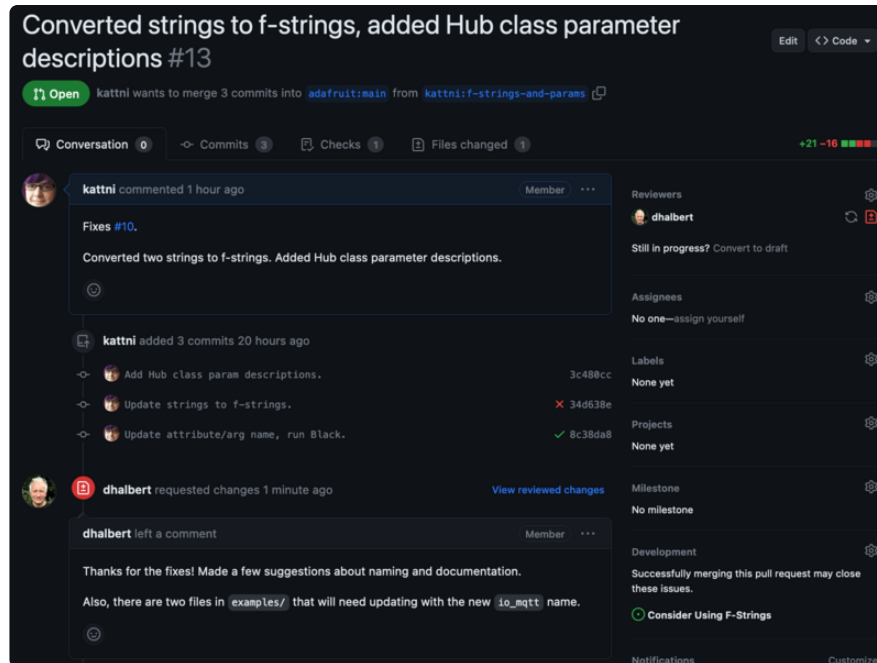
Code reviews are an integral part of open source software development, and software development in general. One of the greatest things about open source software is community involvement. A **review** is an opportunity for someone with different ideas and knowledge than you to take a look at your code, and either verify it's good to go, or identify places where it can be improved. This applies to beginners and experts alike; everyone can benefit from a second set of eyes on their code.

Reviews should be a positive experience, regardless of the outcome. All feedback should be constructive and positive. Keep in mind, all feedback provided is regarding your code, not you as a contributor. Everyone involved needs to be receptive to feedback and willing to participate in the conversation.

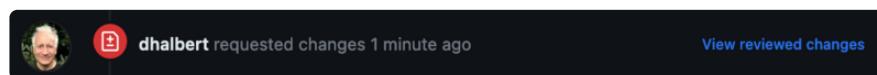
Remember to be patient; sometimes it can take a bit for someone to review your code. We regularly check on open PRs and often request a review from our

CircuitPython Librarians review team. However, a review request does not guarantee an immediate review. Many of the folks on this team are members of the community volunteering their time.

I've waited a bit, and I received an email saying there was an update to my PR. Time to take a look!

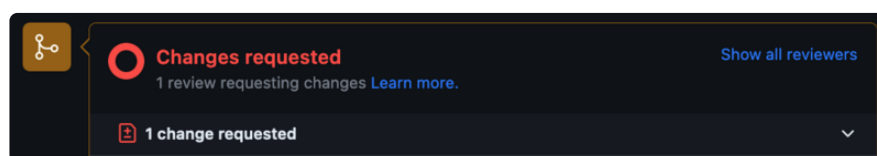


Below the commit list, there's a new section, beginning with "**dhalbert requested changes 1 minute ago**".



Dan, @dhalbert on GitHub, has reviewed my PR! Thanks, Dan!

Remember, you can always check the status of the PR to see where things are. The status of this PR is now **Changes requested**.



Some PRs are ready to go on the first try, and will be merged immediately. If this happens, excellent! You can skip the next section to find out how to continue.

Others require a bit of work first, as with my PR. The next section will take a look at what happens when changes are requested.

The Change Request

Many pull requests are not accepted in the state that they are initially submitted, especially as they get more complex. When a reviewer finds something that needs fixing or could be improved, they will submit a **change request**. This is, as it sounds, a request for you to make changes to your contribution.

GitHub makes this process seamless by allowing a reviewer to go through your code line by line and make suggestions exactly where the change needs to be made. This means you don't have to hunt through your code to figure out what the reviewer was referring to. It's important to read through the entire thing so you don't miss any of it.

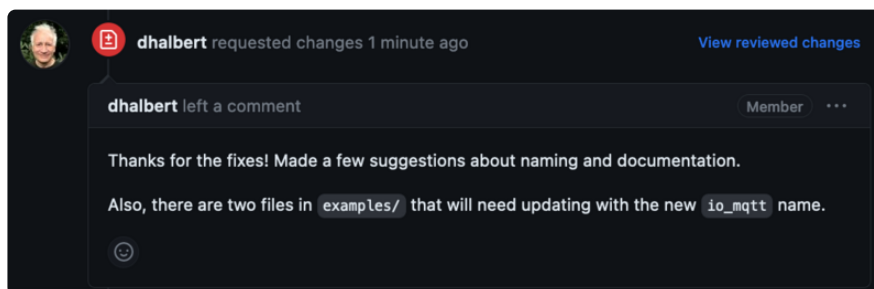
The change request can be viewed in a couple of different ways. The first will show up as a series of comments within the Conversation tab of your PR. The second shows the comments in your code under the Files Changed tab. There's no right or wrong way to view your change request; you should use the method that works best for you. Let's take a look!

The Conversation View

The Conversation tab is often where you start when viewing your PR. A change request will show up in this tab as a series of nested comments, including code snippets. There are a few sections within the request: the main review comment, the nested review comments, and reply or resolve.

Main Review Comment

This is the **main review comment**. Sometimes this contains only a general comment, and the important bits are below it. In this case, though, Dan has mentioned that there are two example files that I also need to update. This is what makes reading this comment so important! The GitHub interface does not allow you to comment on files not included in a pull request, and therefore, if there are other files that need updating, the best you can do is mention them in a comment. Always read through what your reviewer has to say.



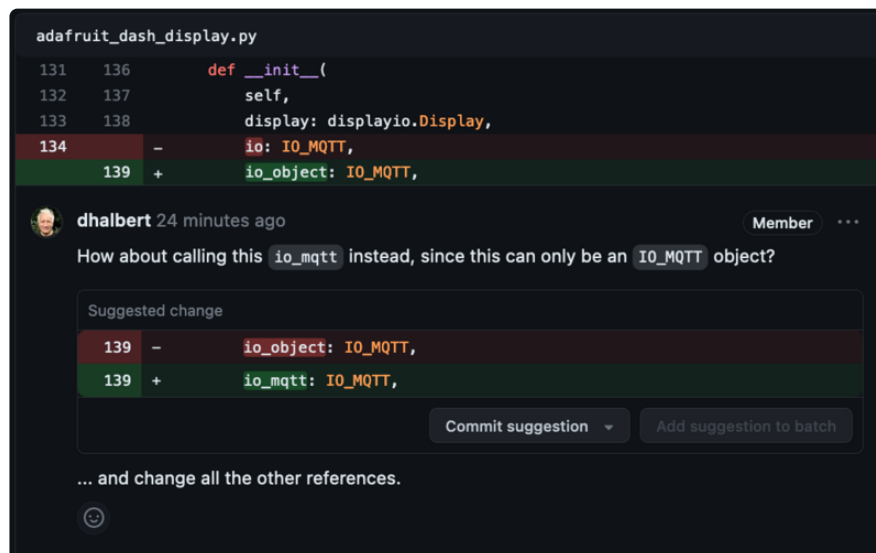
Nested Review Comments

Below the main comment, you'll find a series of nested comments that include code snippets from your pull request. They show the original code and your changes, color coded as in the other comparisons you've seen, and the associated review comments.

Dan's first suggested code change is a more precise name, and he has explained why the suggested name would be a better option.

He also included the **Suggested change** using the GitHub interface, so I can clearly see exactly what he means. Not all reviews will include this.

As there are many references to this new name, Dan also included a note about updating all of the references to it in the code.



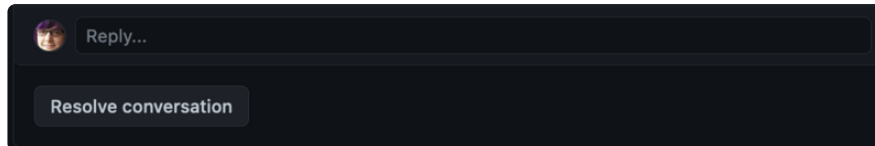
Reply or Resolve

At the bottom of each nested comment, you'll find the option to **Reply...**, or **Resolve conversation**.

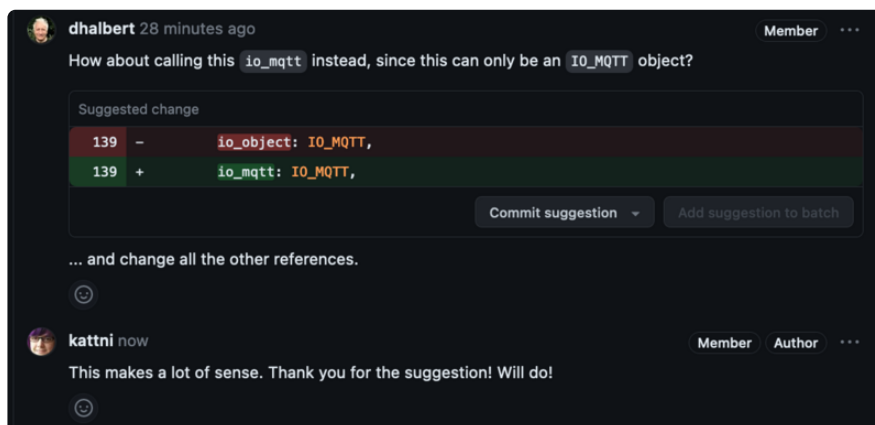
The **Resolve conversation** button is there for you and the reviewer. We use it in two ways. One is for the PR author to click when the update has been applied and pushed to the PR, and the author is satisfied that the requirements have been met. Two is for the reviewer to click when they feel that their requested changes have been made, if the author has opted not to click it. Don't spend a lot of time worrying about whether you've done everything right before clicking; GitHub makes it simple to "unresolve" a conversation in the event that more is needed! Note that the usage of this button may differ based on the specific project to which you are contributing.

There will come a time when you receive a suggestion in a review that doesn't make sense or you don't agree with. You have every right to ask questions or discuss any

part of a review. You can reply by clicking in the **Reply...** dialogue and typing your response. Pull requests are setup to handle forum-like discussions. Feel free to ask for clarification, explain the reason you chose to do something, simply thank someone for their assistance, or open any form of discussion you feel is needed for your review. Some more involved PRs have extremely lengthy discussions as code goes through multiple iterations and changes. This is great! You should always feel comfortable continuing the discussion if you feel it's necessary. We definitely do!



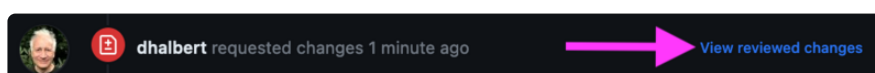
I decided to let Dan know that I agreed with his suggestion and will be making the change. My reply was sent to this specific change, so it is nested along with it. If Dan replied, the thread would continue as seen here. That way, if there are questions about multiple changes, they don't get crossed!



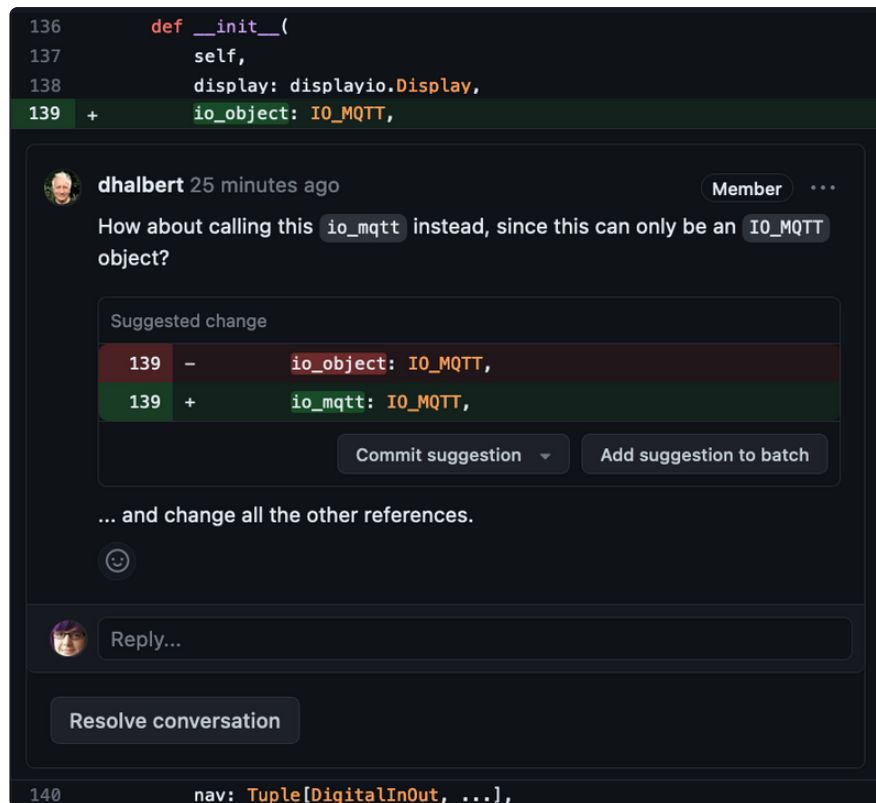
The Files Changed View

Requested changes in the Files changed tab show up in a very similar way, however, here they are shown within the updated file. This view is sometimes easier for folks because it provides a little more code context to the requested changes, than simply viewing the code snippets alone.

You can get there by clicking the **Files changed** tab or by clicking **View requested changes** at the top of the review in the Conversation tab.



Here you'll find the same review comment structure. Instead of only including the code snippet, it is nested within the overall file comparison. This can sometimes make it more obvious where in your code the change is being requested.



I agree with Dan's suggestions, so I'm going to go ahead and make the requested changes.

Submitting the Requested Changes

Submitting fixes for a change request can be done in two ways: using Git, and using the GitHub interface. However, there are many situations where the GitHub interface is not an option. This section covers each one, and explains the benefits of and caveats involved with both.

Using Git to Commit Suggested Changes

The process for using Git to commit suggested changes to a pull request is almost entirely identical to the process used to handle resolving the failed checks when the PR was created. One of the benefits to this is that you're already using a Git/GitHub workflow, so it may make sense to simply continue along that path. It also means that your local copy remains up to date with the remote copy on GitHub. Further, understanding how to use Git to submit changes to a PR is important; if you are asked to make changes to files not already included in the PR, you will need to use Git. So, it's convenient that the process is a familiar one.

Remember, **pushing a commit to the current branch will update the open PR**. This is valid for the entire duration of an open PR, regardless of the reason for submitting updates. Therefore, following the process you've already learned will get you the results you're looking for.

There were a few elements to the change request, including a rename, and updating two example files. As those files were not included in the initial PR, they'll need to be submitted using Git. I've renamed the argument/attribute `io_object` to `io_mqtt` in the `adafruit_dash_display.py` file. I have run my series of `git` commands: `status`, `add`, `status`, `commit`, `status`, `push`.

```
kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params *! (9
> git status
On branch f-strings-and-params
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   adafruit_dash_display.py

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

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params *! (9
> git add adafruit_dash_display.py

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params ** (9
> git status
On branch f-strings-and-params
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   adafruit_dash_display.py

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params ** (9
> git commit -m "Update name per review."
[f-strings-and-params f9a006e] Update name per review.
 1 file changed, 12 insertions(+), 12 deletions(-)

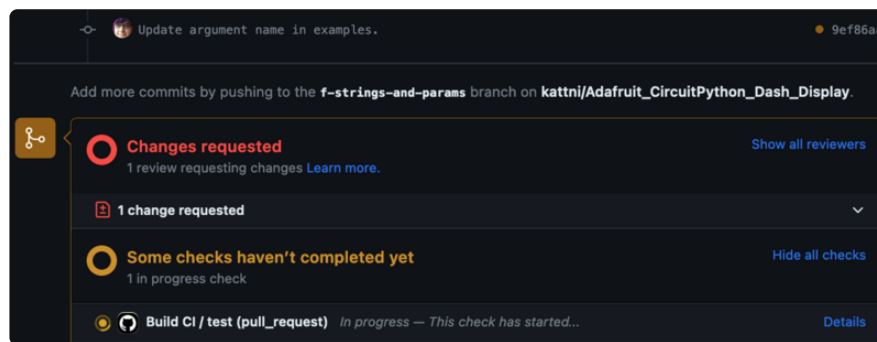
kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params (f9a0
> git status
On branch f-strings-and-params
nothing to commit, working tree clean

kattni:Adafruit_CircuitPython_Dash_Display | f-strings-and-params (f9a0
> git push kattni f-strings-and-params
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 369 bytes | 369.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_Dash_Display.git
 9ef86aa..f9a006e f-strings-and-params -> f-strings-and-params
```

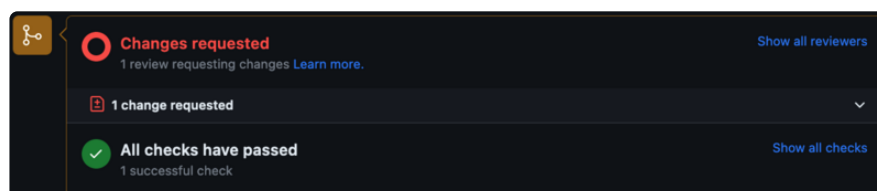
Excellent! This commit will now show up in the pull request. But, I have more changes to make. So, I'll make the changes to the example files, and then run the exact same series of `git` commands to commit my second set of changes to the PR.

Now it's time to check on my latest commit and see where the PR is at by looking at the status. There is a yellow dot next to the commit hash in the commit list indicating checks are occurring. The overall status is still Changes requested, because the

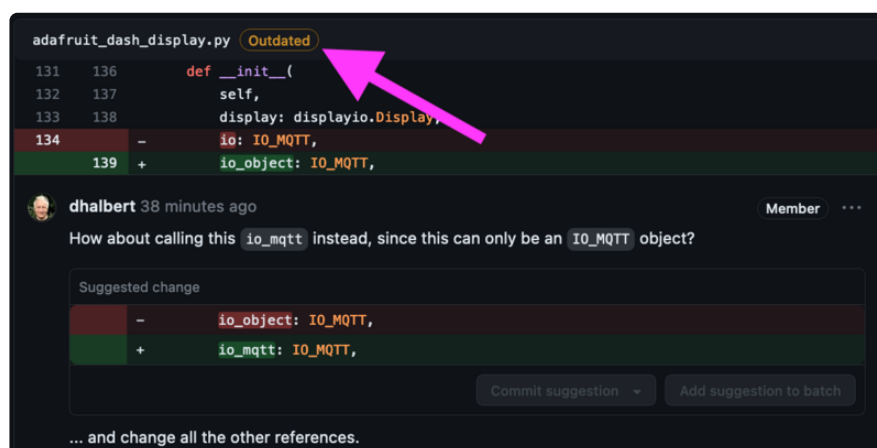
changes have not yet been reviewed. As you can see, the checks are, once again, in progress.



You'll want to give the checks a chance to complete. If they fail, follow the process outlined on the previous page in this guide to resolve the issue. In this case, the checks have passed.



You can now scroll up to the first requested change, and you'll see that it has been marked as **Outdated**. This indicates that a commit has occurred that altered the specific line (or lines) of code addressed in that specific review comment. This does not mean that you've completed the change to the satisfaction of the reviewer, it simply means GitHub sees a change in that spot, and is noting that the existing review no longer applies.



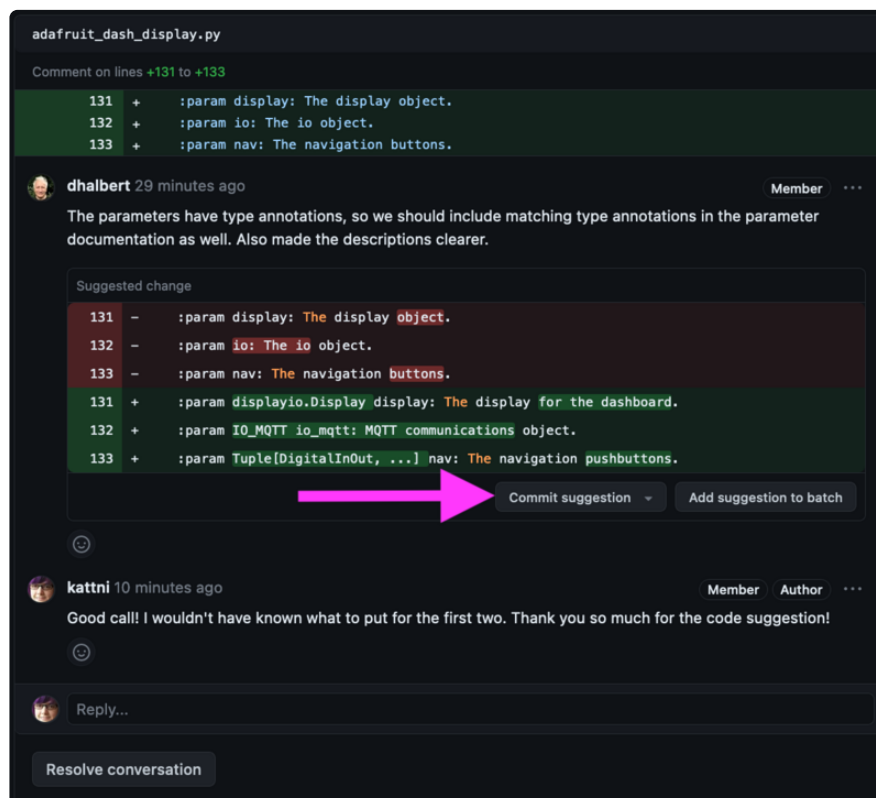
There is one more change to address. This one we'll do using the GitHub interface.

Using GitHub to Commit Suggested Changes

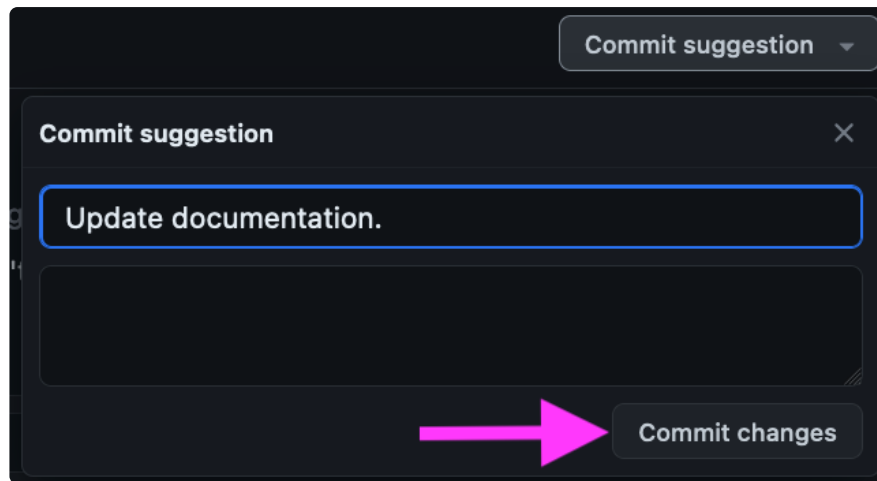
GitHub includes a feature that allows you to accept suggested changes directly from a review comment, and commit them directly. This is a super convenient way to

ensure that you're submitting the change exactly as requested by the reviewer. However, it is only available when the reviewer has provided a suggested change within a review comment. This means it is not possible to use this process to submit changes to files not already included in your PR. Further, it will result in your local copy of the code no longer being up to date with the remote copy. It's one simple step to keep up with this, but if you forget and push more changes, it can get hairy. You can use both Git and GitHub on the same PR. For my PR, I used Git to fix the naming suggestions. I'll use the GitHub interface for the documentation suggestions.

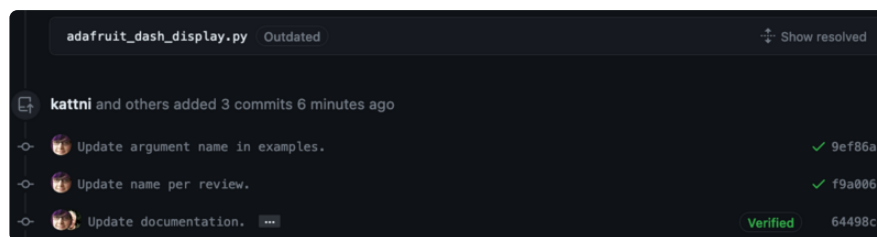
To begin, navigate to the specific change you want to commit. You can do this through either the Conversation or Files changed views. Click **Commit suggestion**.



The resulting dialogue comes with a prepopulated commit message, which will always be "Update filename.py", where the name and extension of the file match the file you are updating. It also provides the option to type in a commit description. You should continue your habit of descriptive commit messages and update the commit message. Once you've done so, click **Commit changes**.



The review comment will be marked as Outdated, and collapsed as completely resolved, because this method guarantees that the update matches exactly what the reviewer was looking for. If you need to see it again for some reason, you can always click "Show resolved" on the right. The new commit shows up in the list, the same as the others.



This is the last of the requested changes! The pull request is ready for a second look. You have a couple of options here. You can leave a comment tagging the person who began the review by including their GitHub username beginning with @, e.g. @dhalbert. You can also simply request another review by clicking the circle-arrows next to the reviewer's name in the Reviewers list at the top of the right column of your PR.



I've let Dan know that I made the changes, and now I'll wait until he's next around to see whether there's anything else to do for this PR.

Don't Let Your Local Code Get Behind

When there are more commits on a copy of code, whether local or remote, you call that copy **ahead**. The copy that does not have those commits is considered **behind**. At this point, **the remote copy of the code is one commit ahead of your local copy**,

because you used GitHub to commit directly to the remote copy. When you are working on an open pull request, it's very important to stay on top of keeping your local copy up to date.

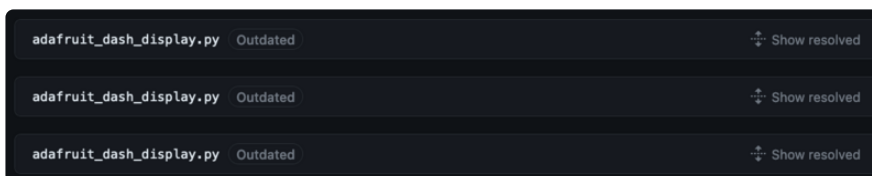
When you end up in this situation, you'll want to complete the following step. Return to your terminal program and run `git pull remotename current-branch-name`, replacing `remotename` with the name of your remote (your GitHub ID), and `current-branch-name` with the name of the branch you're using with the PR.

```
> git pull kattni f-strings-and-params
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 3.82 KiB | 3.82 MiB/s, done.
From github.com:kattni/Adafruit_CircuitPython_Dash_Display
* branch          f-strings-and-params -> FETCH_HEAD
  f9a006e..64498ce f-strings-and-params -> kattni/f-strings-and-params
Updating f9a006e..64498ce
Fast-forward
 adafruit_dash_display.py | 6 +++---
 1 file changed, 3 insertions(+), 3 deletions(-)
```

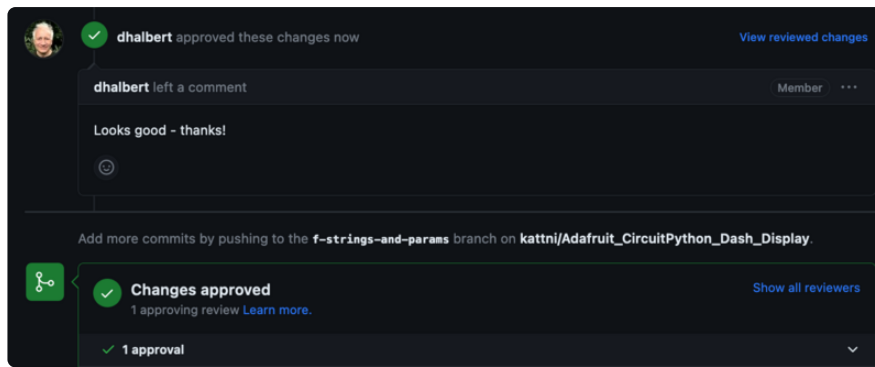
The local copy of the PR branch is now up to date with the remote copy. That way, if any further changes are requested that require using Git, it's ready to go.

Changes Approved

I received an email letting me know there was an update to my pull request. Dan's back, and he's taken another look. He is happy with the changes I made for the first two request comments, and has resolved both of those as well.



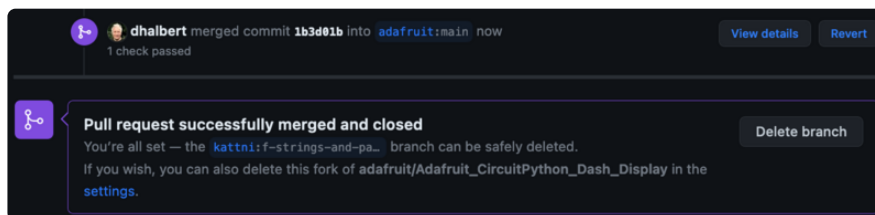
My PR has been approved! Dan has reviewed the changes I made following his change request and concluded that I've made them to his satisfaction. He comments to let me know this is the case, and approves the changes.



Merge

The final step in this process is having your pull request merged. A **merge** takes the changes submitted in the PR, and integrates them into the original copy of the code. Only the final commit is merged, as the final commit already includes all of the changes made in the previous commits.

Dan merged my contribution! My updates are now part of the Adafruit version of the Dash Display library, and moving forward, others may use and enjoy them.



Congratulations! You've successfully had a PR reviewed and merged. Now you're ready to pick out another issue to address, and continue your journey through open source software contributing!

Post-Merge Cleanup

Keeping Branches Trimmed outlines a couple of things you can do post-merge to help keep things tidy for yourself and other library contributors.

Staying Up To Date

Congratulations! Your pull request was approved and merged, and your code is now part of the original project's repo. This means that your fork's main branch is now behind the original project's main branch. It's now time to update your main to match the original project.

The first thing you want to do is return to the main branch. If you're unsure which branch you're currently on, type `git branch` to see a list. The current branch will be

highlighted with an asterisk next to it. You should still be in the branch you created. Now, let's return to main.

To check out the main branch, enter the following `checkout` command:

```
git checkout main
```

```
1486 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git checkout main
Switched to branch 'main'
Your branch is up to date with 'kattni/main'.
```

Next, we're going to utilise the original project remote we created. To get the updates from the remote repo, we're going to use `fetch`. Remember, `fetch` grabs the the newest version of the remote repo, but does not merge it into the current repo.

Remember, you named the original project's remote repo with the owner's GitHub ID. You'll use this name when you merge the two main branches together. Since I cloned an Adafruit repo, I'll be using `adafruit`.

To fetch the updated remote, enter the following `fetch` command, replacing `ownerid` with the name you assigned to the remote repo:

```
git fetch ownerid
```

```
1487 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git fetch adafruit
remote: Counting objects: 1, done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), done.
From github.com:adafruit/Adafruit_CircuitPython_CircuitPlayground
 862a379..d0aa6dc  main      -> adafruit/main
```

Now we're going to `merge` the current data into our local repo. Remember, a merge takes the information from one branch and combines it into another. In this case, it's going to take the current version of main from the remote repo and combine it with the main branch on your local repo. This will bring you even with the remote main, including the changes you submitted.

To merge the remote main with your main, run the following `merge` command, replacing `ownerid` with the name you assigned to the remote repo:

```
git merge ownerid/main
```

```
1488 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git merge adafruit/main
Updating 862a379..d0aa6dc
Fast-forward
 adafruit_circuitplayground/express.py | 21 ++++++-----
 1 file changed, 12 insertions(+), 9 deletions(-)
```

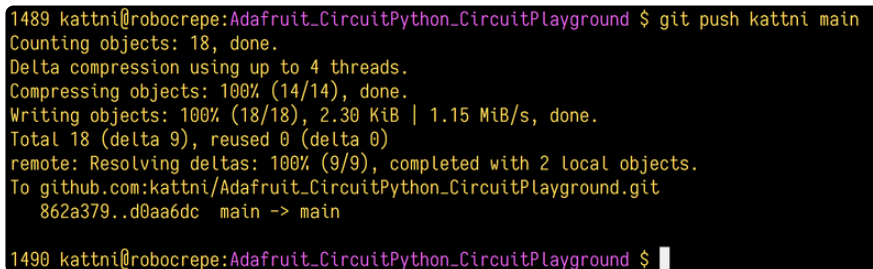
Those numbers may look familiar. They match the changes from my PR! This will not always be the case. With larger projects, people are constantly submitting changes and the list from this step may be lengthy. Regardless, you're set to move on to the next step - the results aren't important to the process of updating.

Now your local repo is even with the remote repo. Your remote fork on GitHub, however, is not. It does not automatically update when you update locally. So, you must manually **push** your locally updated main to your remote fork. This uses the exact same command format as pushing your working branch did. This time, however, we're pushing the main branch.

Remember, you named your remote repo with your GitHub ID. You'll use this to **push** the updated main branch in the same way you did when pushing your working branch.

To update your remote fork on GitHub, type the following **push** command, replacing **yourID** with your GitHub ID:

```
git push yourid main
```

A terminal window with a black background and yellow text. The prompt is '1489 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground \$'. The command 'git push kattni main' has been executed. The output shows the progress of pushing the main branch to the remote repository, including object counting, compression, and resolution of deltas. The final line of the output is '862a379..d0aa6dc main -> main'. The prompt for the next command is '1490 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground \$' with a cursor at the end.

```
1489 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git push kattni main
Counting objects: 18, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (14/14), done.
Writing objects: 100% (18/18), 2.30 KiB | 1.15 MiB/s, done.
Total 18 (delta 9), reused 0 (delta 0)
remote: Resolving deltas: 100% (9/9), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_CircuitPlayground.git
   862a379..d0aa6dc  main -> main
1490 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

Now the main branch on both your local repo and your remote repo are up to date. You're ready to continue working. From here, you can return to your previous branch and update it, or you can create a new branch and start on a new contribution.

Keep in mind, if you step away from a repo for a period of time, you should always update it before creating a branch to work with or you may be working with out of date data. This can lead to conflicts when attempting to merge later. Conflicts can be incredibly frustrating, but are easily avoided if you keep your branches up to date as you go. When it comes time to create your PR, verify that it can be merged automatically before creating it. If it can't, you may have been working with an out of date branch and will need to update it before creating the PR. Don't be afraid to ask for help with this! Sometimes it's a simple fix, other times it's more complicated. We're always happy to help you work through it.

Keeping Branches Trimmed

The more you contribute, the more branches you'll create. It never seems like it when you create your first branch, but eventually you're going to have a lengthy list of branches. You it's simple to avoid this by deleting your branches as you go.

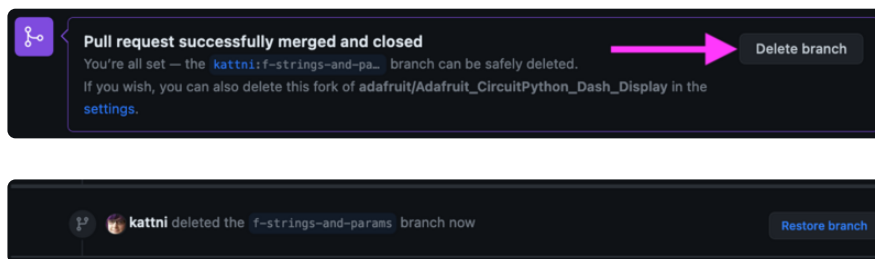
You must be certain you're ready to delete your branch before deleting it. Deleting your branch does not delete your work AS LONG AS YOUR WORK HAS BEEN MERGED. If you have unmerged work on your branch, DO NOT DELETE IT.

Do not delete your branch if you have changes that have not been merged.
YOU WILL LOSE THOSE CHANGES.

Keep in mind that you have two locations from which branches require deletion: remote and local. Your remote branches are located on GitHub once you push them. Your local branches are located on your computer when you create them.

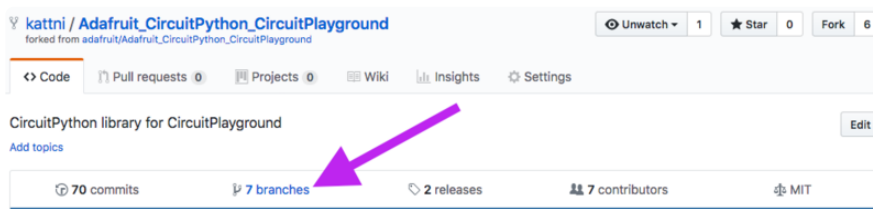
Deleting Your Remote Branch

After your PR is merged, there is typically a notification at the bottom that tells you it's now safe to delete your branch, with a button to delete it. If you click this button, it will delete the branch from GitHub, but not from your local copy.

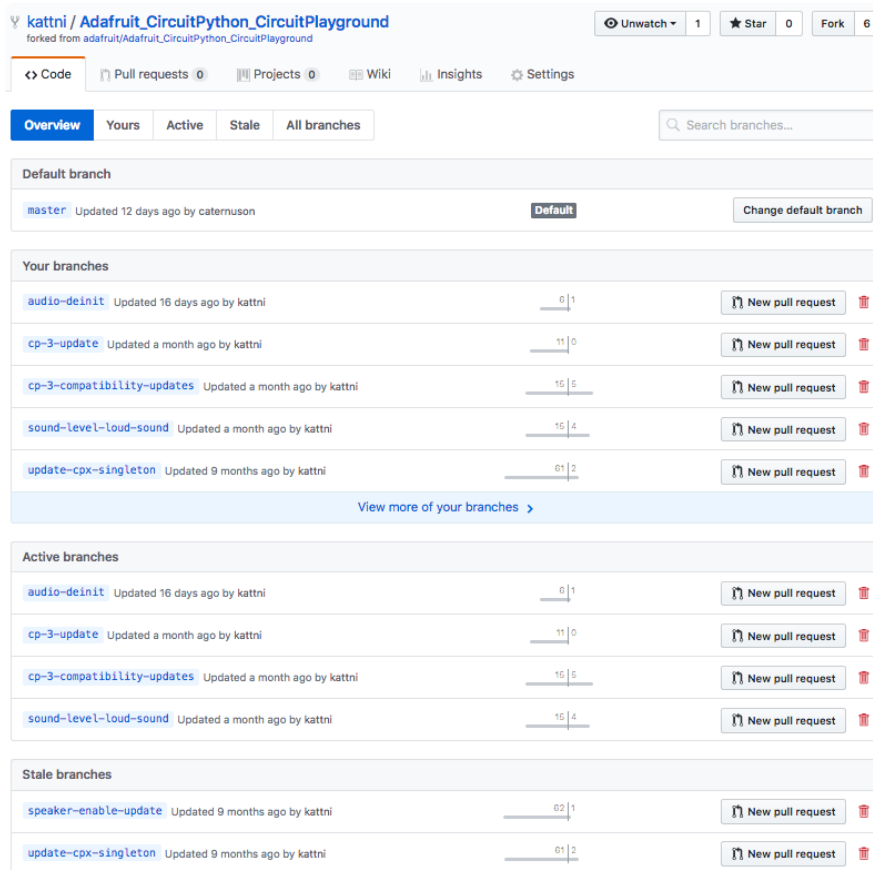


As you can see, when you use the link to delete it, it gives you the option to restore it.

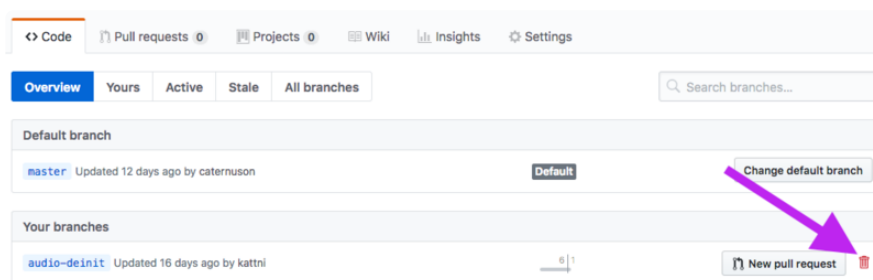
If you're not around when your branch is merged, you can still delete your remote branch from GitHub. Navigate to the main page of your repo, and locate the **# branches** tab, where # is the number of branches you currently have on your repo. In my case, it's **7**. Click the link to the tab.



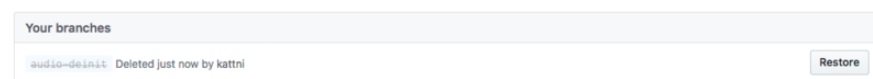
Here you'll find multiple lists of branches. The first is **Your branches**, which contains every branch you have on your repo. The second is **Active branches**, which contains branches that were recently updated. The third, if you haven't kept up with deleting branches as you go, is **Stale branches**, which contains a list of branches that haven't been updated in a while.



You can delete from any of these lists by clicking the **trash bin icon delete button** located on the right side on the same line as the branch name. I would like to delete my most recent branch, so I'm going to delete it from **Your branches** by clicking the **trash bin icon**.



Once you delete the branch, you'll have an opportunity to restore it by clicking the restore button that appears. This line will disappear from your page once you refresh, so be certain you meant to delete that branch before leaving the page.



Now you've deleted your remote branch, but it's still in your local repo. Next, we'll go over how to delete a branch from your local repo.

Deleting Your Local Branch

Open your terminal program and navigate to your repo folder. You'll want to make sure you're on the main branch, so first run the following `checkout` command:

```
git checkout main
```

```
1486 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git checkout main
Switched to branch 'main'
Your branch is up to date with 'kattni/main'.
```

If you'd like to see a list of your current local branches, you can run `git branch`.

Next, you'll want to delete your branch. You can use tab-completion to complete your branch name. To do this, start typing the beginning of the name at the end of the command, and then hit tab.

To delete your branch, run the `branch` delete command, replacing `your-branch-name` with the name of the branch you're deleting:

```
git branch -d your-branch-name
```

```
1490 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git branch -d play-file-stop-tone-fix
Deleted branch play-file-stop-tone-fix (was c70fe23).
1491 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

Note that in some circumstances, this command will not delete your branch. One situation where it will fail is if you have changes that haven't been merged. Consider the command provided as a way to be sure you don't delete any unmerged changes.

If you are CERTAIN that you do not want to keep the changes in a given branch, you can run the same command with `-D`, and it will force deletion of the branch. However, sticking with `-d` ensures that you do not accidentally delete unmerged changes.

That's it! You're all set. You've deleted your branch locally and remotely. You're ready to create a new branch and get started on your next contribution!

Giving a Review

Reviews are a crucial part of the contribution process. They can also be a choke point in the process, as often the number of people available to do reviews is significantly less than the number of contributors. Our solution is to accept reviews from a wider range of people. Even if you don't have write access to a repo, you're always welcome to provide a review on a pull request.

You may be saying to yourself, "But, I'm new to all of this, what do I have to offer as a reviewer?" Everyone has something to offer! Pull requests can consist of everything from simple typo fixes to massive core changes, and every single one of them has the potential for bugs. Bugs can be anything from the code not working at all to a typo in a docstring that the linter didn't catch. Even if you aren't an experienced programmer, you have what it takes to be a capable reviewer.

Don't worry about approving a PR you don't entirely understand. Simply be clear regarding what you checked. For example, if you checked it for typos, include that in the comment section. "Checked for typos. Looks good!" If you tested the code on your own board to make sure it works, let us know. "Verified this works on the correct sensor." This allows you to review the parts you do understand, while giving us the opportunity to check the part of the PR outside your scope.

A Positive Experience

Many fear the review process because they've had a negative experience receiving a review in the past. We strive to mitigate this by perpetuating a positive experience through positive, constructive feedback. We always thank people for their work before providing feedback. Much of the work contributed to open source projects is done so by members of the community. The most important things you can do is make sure those people feel valued as contributors, and help build their confidence. This is an essential part of how we operate and we expect anyone else who participates to do the same. Always consider how you would feel receiving the review you're about to give and make sure that it's absolutely positive about it.

Any feedback can be positive, constructive feedback. It's entirely in how you present it. Simply telling someone, "You're wrong," isn't positive or constructive. The same information can be presented by saying, "Thanks for doing this! I have a suggestion. The change on line 17 could be done differently," followed by your suggested change. Now you've started a conversation. You've provided a review that gives the person a place to start from for improving their changes, and helped create an environment where they can feel confident. This is the most important part of the review process.

This section walks through the steps of giving a review. In this review, I'll be requesting changes to the code and then verifying those changes were made before approving. Let's get started!

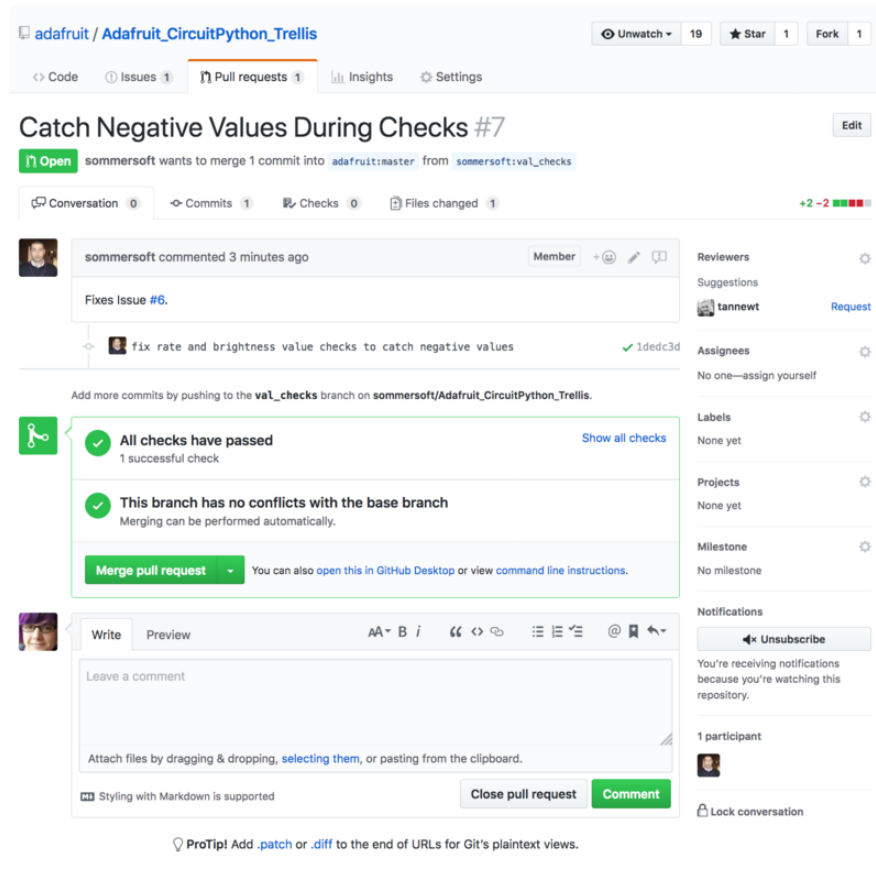
Someone Opened a Pull Request

I've been paying attention to a few repos and I see that Sommersoft opened a PR. Excellent! That means it's time for a review.

The first thing you want to do is wait for GitHub Actions to finish testing. If Actions fails, then the person who opened the PR will need to submit the fixes for that. While

you're welcome to begin the review with suggesting GitHub Actions fixes, it's usually a better idea to give the person a chance to take care of the issues first.

For a detailed look at the elements of a PR, check out the Pull Request Explored section of Open Pull Request.



In this PR, Travis builds successfully and all of the checks pass. Now I can start my review.

Begin Your Review

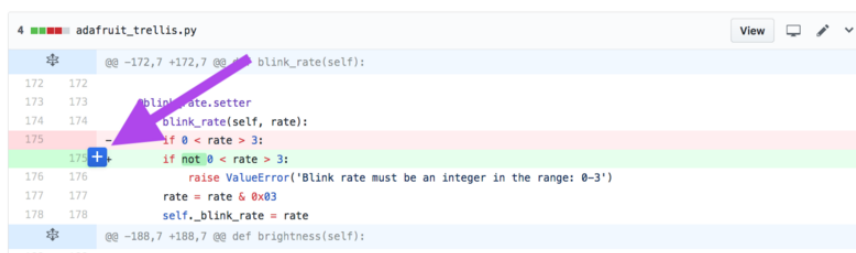
Reviewing begins with looking at the changes included in the PR. Click on the **Files Changed** tab under the title and status of the PR. This will take you to the diff of all of the submitted changes.

```
4 adafruit_trellis.py
@@ -172,7 +172,7 @@ def blink_rate(self):
172 172     @blink_rate.setter
173 173     def blink_rate(self, rate):
174 174         if 0 < rate > 3:
175 -         if not 0 < rate > 3:
176 +         if not 0 < rate > 3:
177         raise ValueError('Blink rate must be an integer in the range: 0-3')
178         rate = rate & 0x03
179         self._blink_rate = rate
@@ -188,7 +188,7 @@ def brightness(self):
188 188     @brightness.setter
189 189     def brightness(self, brightness):
190 190         if 0 < brightness > 15:
191 -         if not 0 < brightness > 15:
192 +         if not 0 < brightness > 15:
193         raise ValueError('Brightness must be an integer in the range: 0-15')
194         brightness = brightness & 0x0F
195         self._brightness = brightness
```

Sommersoft has added a **not** to line 175 and 191. I see some problems, however. It seems like the code will only run if the values are outside what the error says are the appropriate values. It's an easy thing to miss. So, I'm going to start a review to let Sommersoft know my thoughts on the PR.

If the issue you find is something applicable to the entire piece of code, you can simply click the **Review Changes** button at the top left, and leave a comment there. However, when you find issues on specific lines, it's good to leave an inline review with comments at the specific points you're referring to. The issues I've found are on specific lines, so I'm going to begin my review with a line-specific comment.

Mouse over any part of the line of code and you'll see a blue plus appear next to the line number.



Click the blue plus and a window will appear below that line of code.

In the comment field, I'm going to write up my first review comment.

It's always possible that there's a reason the contributor chose to make the submitted changes. If you're unsure, ask about the change. A review doesn't have to be suggested changes, it can simply be a question about why a change was made.

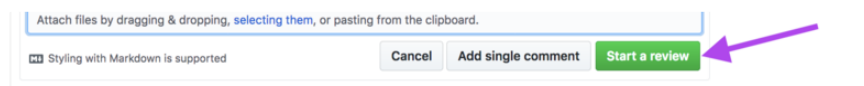
I'm going to ask about the acceptable values. In this case, I'm certain that it's not right as it is, and I have an idea of how to fix it. So, I'm also going to suggest a change to

the code. GitHub supports Markdown in comments, so I'm going to format the code so the comment is easier to read.

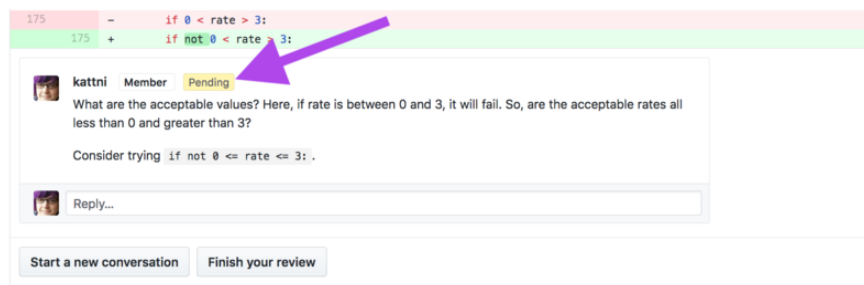
If you'd like to know more about Markdown, click the link in **Styling with Markdown is supported** below the comment field for details.



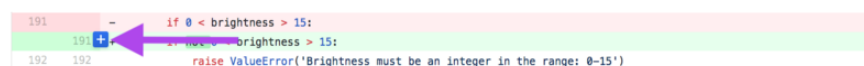
Once I'm done, I'm going to click the **Start a review** button below the comment field.



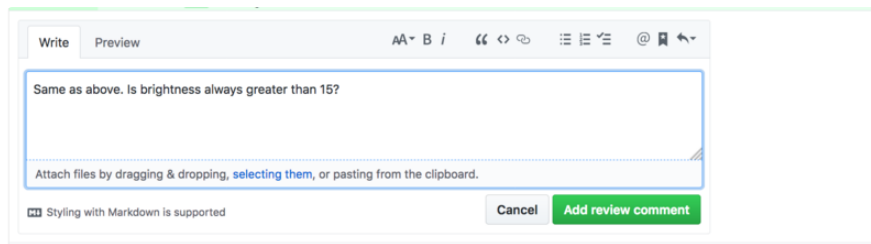
This begins my review by saving this first comment. It won't officially submit it until I'm ready. It will instead show my comment as **Pending** until I choose to complete the review.



Now I'm going to repeat the process for the second issue. First mouse over any part of the line to bring up the blue plus sign.



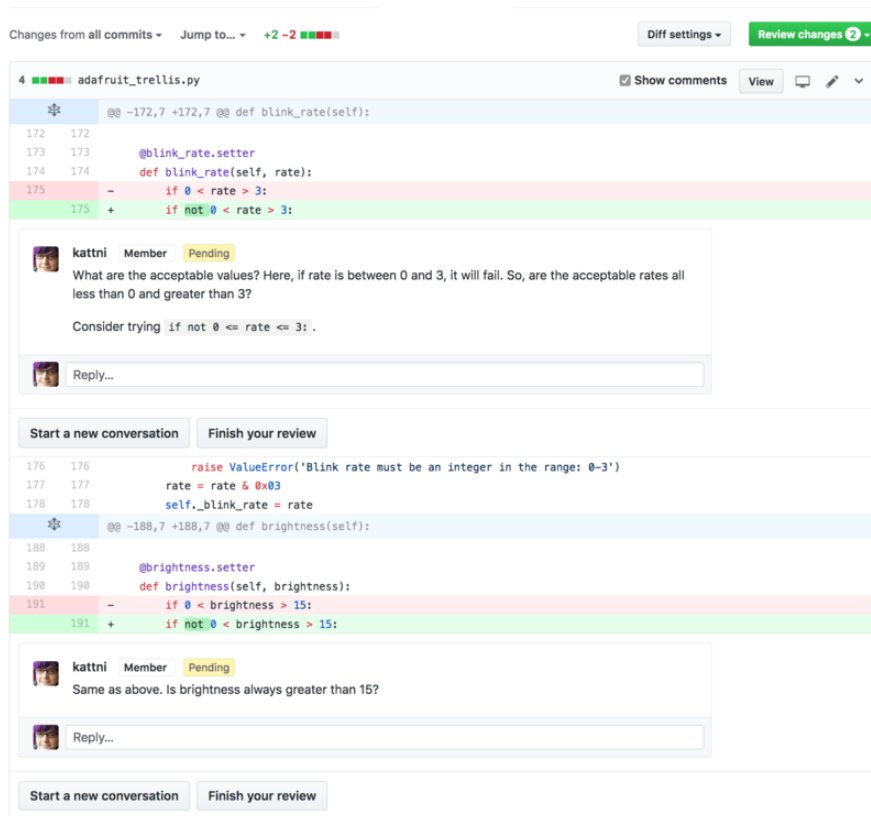
Click the blue plus to bring up the review comment window. Fill in your comment. In this case, since the issue is the same concept as the first, my comment is shorter because I can refer to the existing comment.



This time, since the review is already in progress, the button has changed. Click the Add review comment button to save the new comment.



Now my second comment is pending as well. These are all the comments I would like to make to specific lines of code, so it's time to finalise my review.

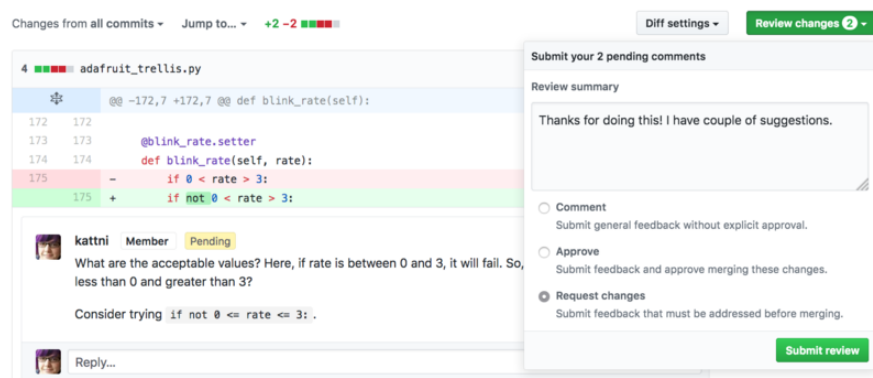


Notice that the **Review changes** button located above the code has a **2** next to it. This is because there are 2 pending review comments. Since I'm ready to finalise my review, I'm going to click the **Review Changes** button.



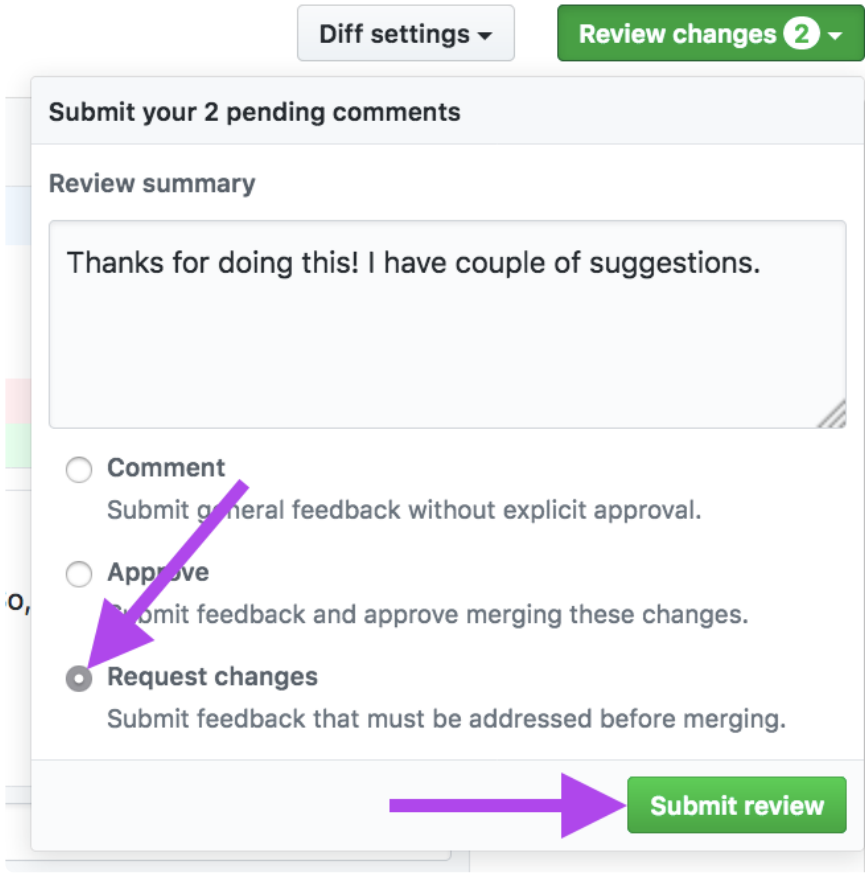
This will open window over the code with a comment field and some options below it. This is where you can add the main comment seen in the review. Whatever you put in this comment field will be at the top of your review, with your in-line comments below

it. Here is where I'm going to thank Sommersoft for the changes he submitted, and let him know that I have a couple of suggestions. Since I outlined these suggestions in the pending comment, I'm not going to reiterate them here.



Notice that there are three options at below the comment box. It defaults to **Comment** which allows you to submit a review containing only a comment if you choose. Below that is **Approve**, which you will choose when you're ready to approve the PR. Last is **Request changes**, which is what I've done in my pending comments, so I'm going to choose it as my option.

Once I choose Request changes, I'm ready to submit my review. So, I click the **Submit review** button at the bottom of the window.



Once I click Submit review, it will automatically take me back to the Conversation tab and show me my review. Since I requested changes, the status at the bottom of the page now reflects that.


The screenshot shows a GitHub pull request interface. At the top, a notification from user 'kattni' states 'kattni requested changes just now' with a 'View changes' button. Below this, a comment from 'kattni' says 'Thanks for doing this! I have couple of suggestions.' The main content area displays two code snippets from 'adafruit_trellis.py'. The first snippet shows a function 'def blink_rate(self, rate):' with a comment '@@ -172,7 +172,7 @@'. The code includes a line 'if 0 < rate > 3:' which has been changed to 'if not 0 < rate > 3:'. The second snippet shows a function 'def brightness(self, brightness):' with a comment '@@ -188,7 +188,7 @@'. The code includes a line 'if 0 < brightness > 15:' which has been changed to 'if not 0 < brightness > 15:'. Below the code, there are two more comments from 'kattni'. The first asks 'What are the acceptable values? Here, if rate is between 0 and 3, it will fail. So, are the acceptable rates all less than 0 and greater than 3?' and suggests 'Consider trying if not 0 <= rate <= 3:'. The second comment says 'Same as above. Is brightness always greater than 15?'. On the right side, there are settings for Labels, Projects, Milestone, Notifications, and a 'Lock conversation' button. At the bottom, a summary bar indicates 'Changes requested' with '1 review requesting changes' and buttons for 'Approve changes' and 'Dismiss review'.

Now I'll be waiting for a response from Sommersoft.

Review Response and Update

I received an email letting me know that there's been a response from Sommersoft on the PR. I'll navigate to the PR again, or refresh the page if I already have it open.


Sommersoft has commented in response to my review comments. He responded to each one individually, so each response shows up below my in-line comments in my review.


 **kattni** requested changes 27 minutes ago [View changes](#)


Thanks for doing this! I have couple of suggestions.

adafruit_trellis.py

```
... @@ -172,7 +172,7 @@ def blink_rate(self):
172 172
173 173     @blink_rate.setter
174 174     def blink_rate(self, rate):
175 -         if 0 < rate > 3:
175 +         if not 0 < rate > 3:
```


 **kattni** 27 minutes ago **Member**
What are the acceptable values? Here, if rate is between 0 and 3, it will fail. So, are the acceptable rates all less than 0 and greater than 3?
Consider trying `if not 0 <= rate <= 3:`.


 **sommersoft** 17 minutes ago **Member**
The acceptable range is 0 to 3. So yes, this would definitely fail that test. Update coming in a few!


 Reply...

adafruit_trellis.py


```
... @@ -188,7 +188,7 @@ def brightness(self):
188 188
189 189     @brightness.setter
190 190     def brightness(self, brightness):
191 -         if 0 < brightness > 15:
191 +         if not 0 < brightness > 15:
```

 **kattni** 27 minutes ago **Member**
Same as above. Is brightness always greater than 15?


 **sommersoft** 17 minutes ago **Member**
Range is 0 to 15.

 Reply...

In his first response, he's answered my question regarding the acceptable values, indicated I was correct with my suggestion, and let me know he will be submitting an update soon.

 **sommersoft** 17 minutes ago **Member**
The acceptable range is 0 to 3. So yes, this would definitely fail that test. Update coming in a few!

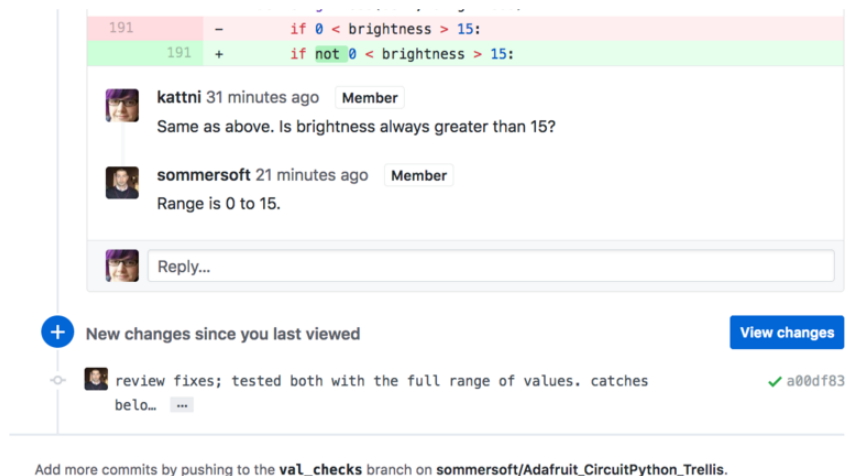
His second comment is similar to mine in that it simply answers the question asked. He's already let me know he will be submitting an update, so he hasn't reiterated it.

 **sommersoft** 17 minutes ago **Member**
Range is 0 to 15.

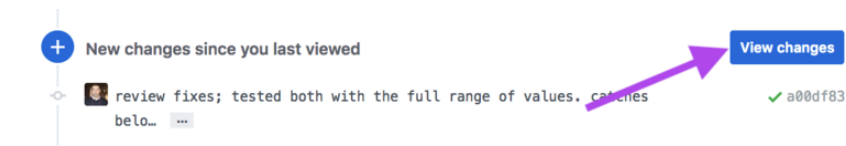
Great! I don't have anything further to add. Sommersoft understands the issues and will be fixing them, so now it's time for me to wait until he submits the fix.

Next, I receive another email letting me know that he's committed a change to the PR. Now I can return to the PR and view the updated changes.

Remember that a PR is visually a timeline. So, the newest commit will show up at the bottom of the list.



Click the **View Changes** button next to the most recent commit to return to the diff.



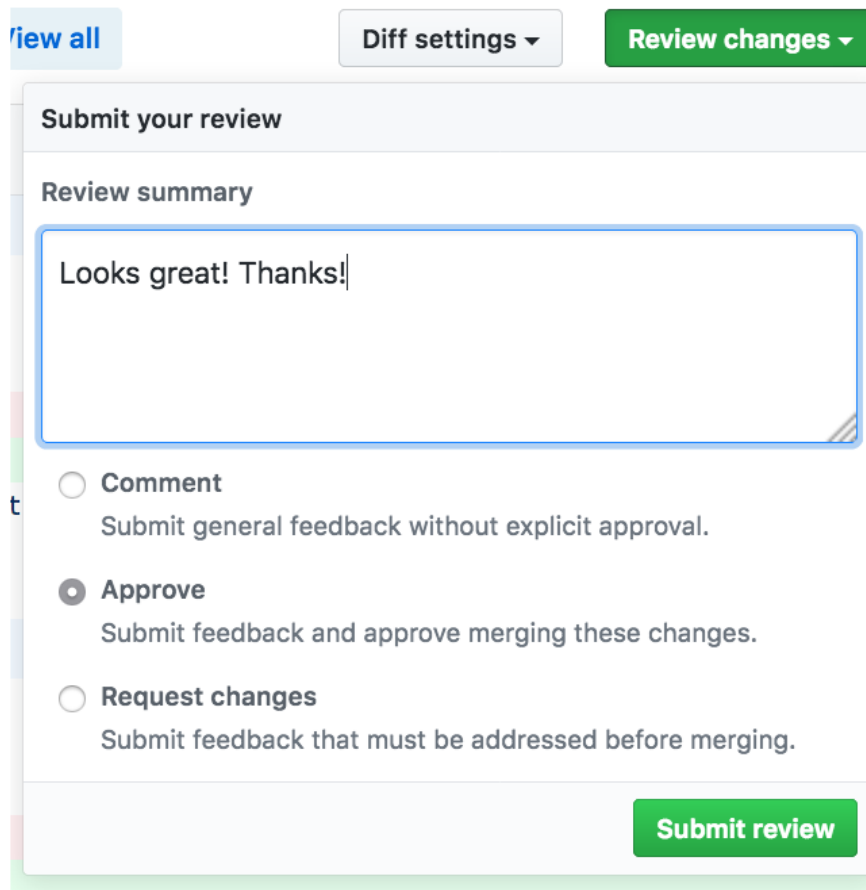
This will take you to the diff showing the most recent changes.

Note the blue information at the top of the diff stating **Viewing a subset of changes**, with a **View all** link. This is because I navigated to the diff using the View changes button next to the most recent commit. In this case, it is in fact all of the changes, however in some cases, the last commit may only affect a small part of the overall PR. This is handy if you requested changes on only a small part of the PR. It allows you to view only the changes you asked for instead of scrolling through a lengthy PR to find what you're looking for. If you wish to see the entire PR in that case, click **View all** and it will return the diff to the original view.



As you can see, Sommersoft took my suggestion for how to resolve the issue with the code. Excellent! I'm ready to approve the PR.

Click on **Review Changes** at the top right of the screen. It will open the same window as before with the comment field and options. This time, I'm going to choose **Approve**, and include a comment to go with it. When I'm ready, I'll click **Submit review**.



View all Diff settings ▼ Review changes ▼

Submit your review

Review summary

Looks great! Thanks!

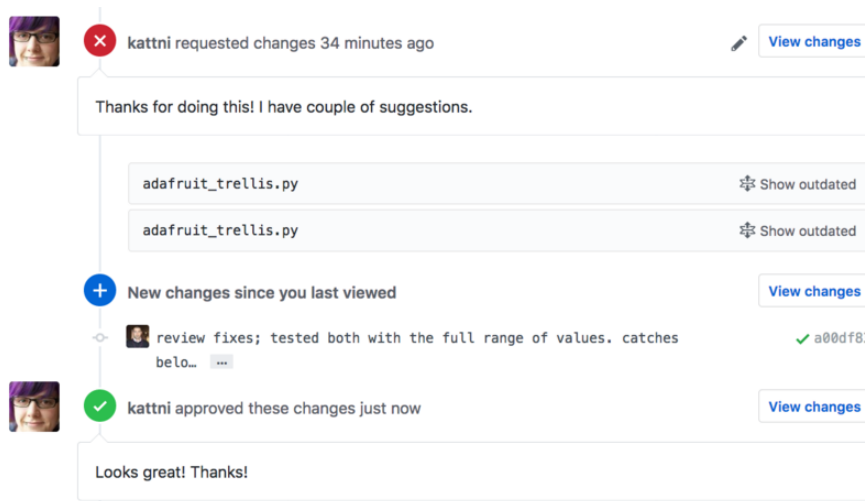
☐ **Comment**
Submit general feedback without explicit approval.

☒ **Approve**
Submit feedback and approve merging these changes.

☐ **Request changes**
Submit feedback that must be addressed before merging.

Submit review

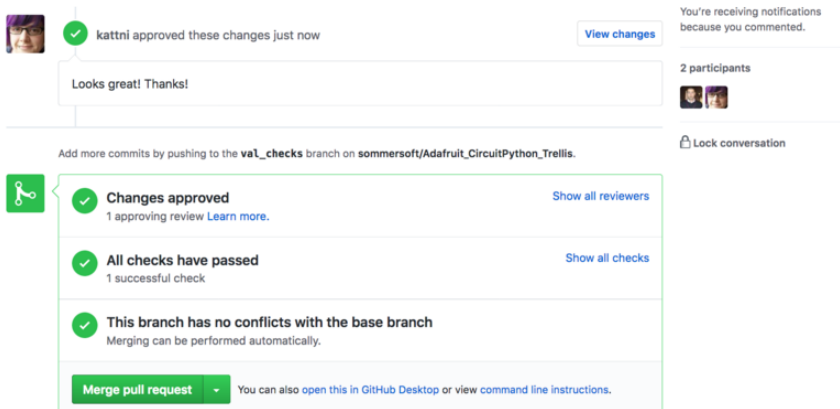
Once I click **Submit review**, it automatically returns to the conversation tab. My change request comments are now collapsed with the option to **Show outdated**. Below the final commit, my approval is listed as such with a green check next to my name.



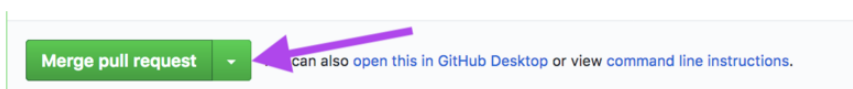
For those without write access to the repo, this is the end of the review process. This completes a significant portion of the entire review and greatly helps those of us with write access. We really appreciate it!

Review Merge

You've reviewed, requested changes, verified and approved the pull request. For those with write access to the repo, the final step is merging the PR. After you submit your review, the PR automatically returns to the conversation tab. You'll see your approval comment, and there should be green checks throughout the status section.

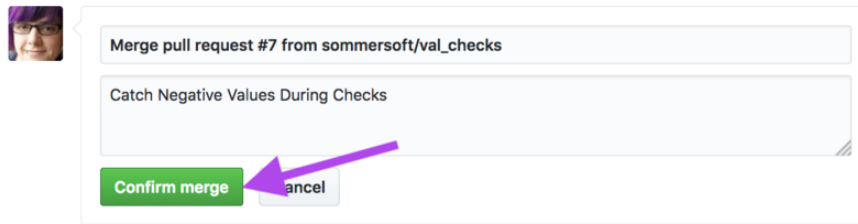


The **Merge pull request** button should also be green. Click the **Merge pull request** button.

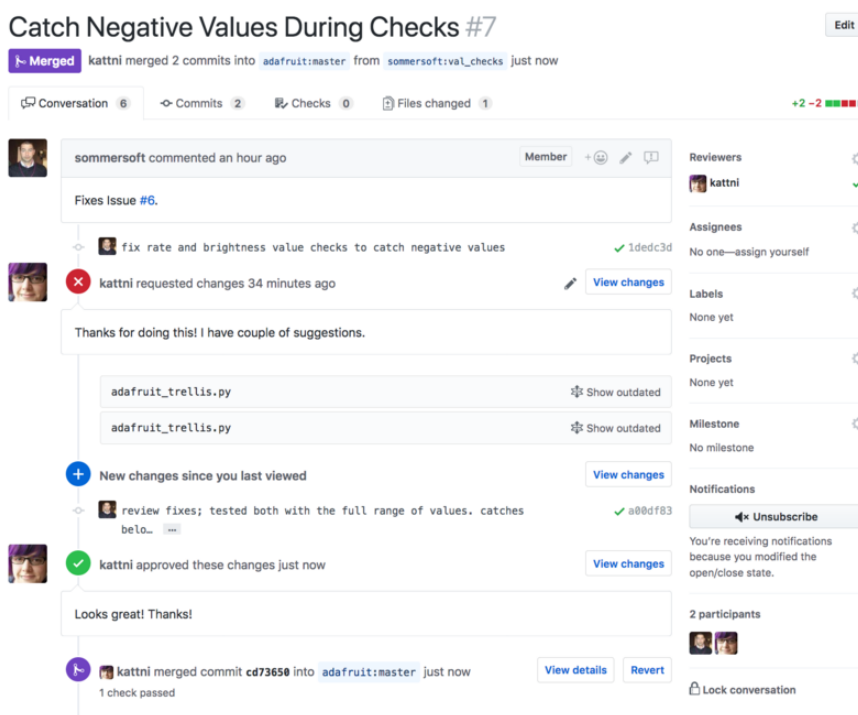


This replaces the entire status section with the confirmation window. There's typically no reason to change any of the populated messages in this window.

Click the green **Confirm merge** button to continue.



Done! The status section at the bottom is immediately replaced with a purple dot followed by the fact that I merged the PR into the project's main branch. It can take a moment for the rest of the PR to catch up - the status at the top may show open for a bit after the merge is complete. Once it's set, the status at the top also showed **Merged** in purple, followed by outlining the number of commits I merged, and the branches merged to and from.



This PR is all set to go! Sommersoft can now do what he wants with his branch and continue on to his next project.

This PR went quickly. As is indicated by the timestamps, it took about an hour. Some PRs can take days or weeks depending on the level of changes or review involved. We look at PRs as a conversation. This means some are submitted early on in the process with the intention of receiving an extended review to help with development. You're welcome to join in on this conversation at any time to provide feedback. We value the contributions of our community members, regardless of what side of the PR they're coming from. Thank you again for being a part of this!

Starting Over Fresh

THESE ACTIONS WILL DELETE ALL WORK NOT MERGED INTO THE ORIGINAL PROJECT INCLUDING ANYTHING YOU COMMITTED AND PUSHED TO ONLY YOUR REMOTE FORK. This will delete your remote fork of the repository, and your local copy of the repository. This means ANYTHING you have not completely PR'd to the original project WILL BE LOST. If you have ANY work that hasn't been included in the original project, SAVE YOUR WORK ELSEWHERE BEFORE CONTINUING.

Sometimes your local repo and remote fork will get so far out of date that it's easier to start over fresh. This is especially useful when the original project makes a major change like updating the default branch. This section will explain how to delete the right things so you can begin again at the same point as the original project.

There are four steps to this process. You'll want to delete two things: your **local clone**, and your **fork**. Following that, you'll fork the repository again, and create a new local clone.

For this section, I'm going to work with the Adafruit CircuitPython Pypixelbuf repository.

Deleting Your Local Clone

The first step is to delete your local clone. Your local clone is the directory into which you cloned the repository in the first place. This action will remove the local copy of the repository from your computer.

You can **use the command line to delete the directory**. First, `cd` to the directory that contains your Git directories, in my case `~/repos/`. Then delete the desired directory by running the following command, replacing `your_repo_directory_name` with the directory you wish to delete:

```
rm -rf your_repo_directory_name
```

```
3643 kattni@robocrepe:~ $ cd repos/
3644 kattni@robocrepe:repos $ rm -rf Adafruit_CircuitPython_Pypixelbuf/
3645 kattni@robocrepe:repos $
```

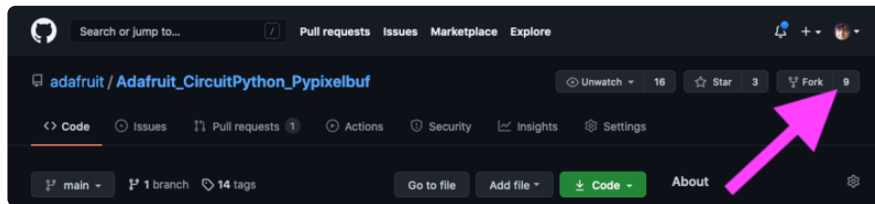
Alternatively, you can use your file explorer to delete the directory.

Now that your local clone is deleted, it's time to delete your remote fork.

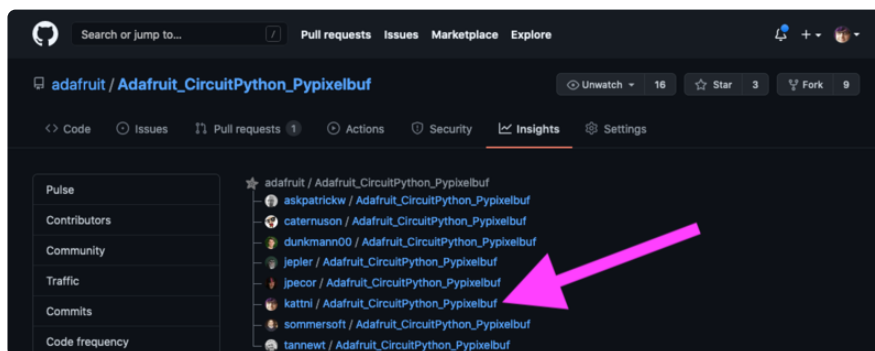
Deleting Your Fork

The next step is to delete your remote fork. This is done through GitHub. In your browser, navigate to your remote fork. You can do this multiple ways. This is one option.

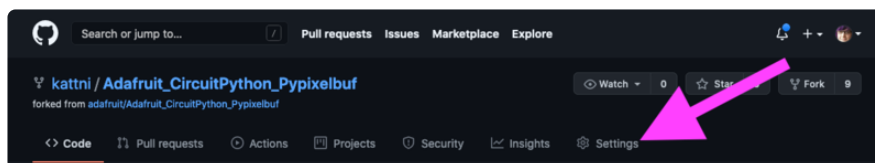
Navigate to the original project repo. Find the Fork button in the upper right corner of the screen. Next to it is a number (the total number of forks made by everyone who has forked this project). **Click the number next to Fork.**



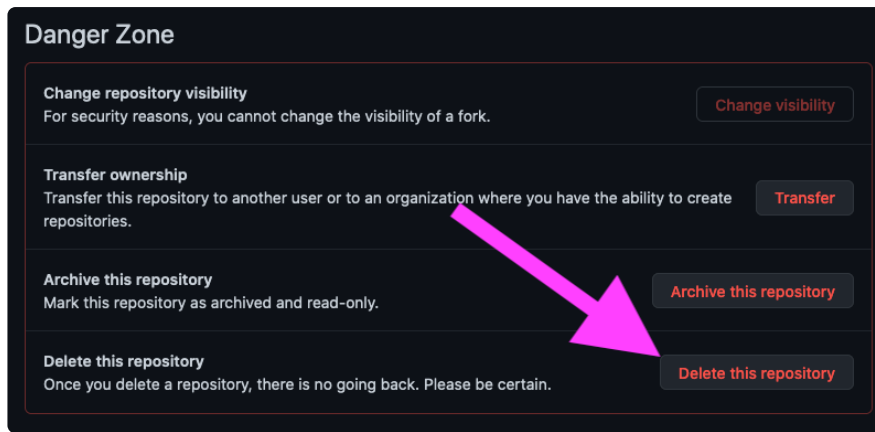
This will take you to a list of all of the forks of that repository, in the form of a list of links. Find your GitHub user ID in the list to find your fork. **Click the link next to your name** to go to your forked repository.



Once in your fork, click **Settings**.

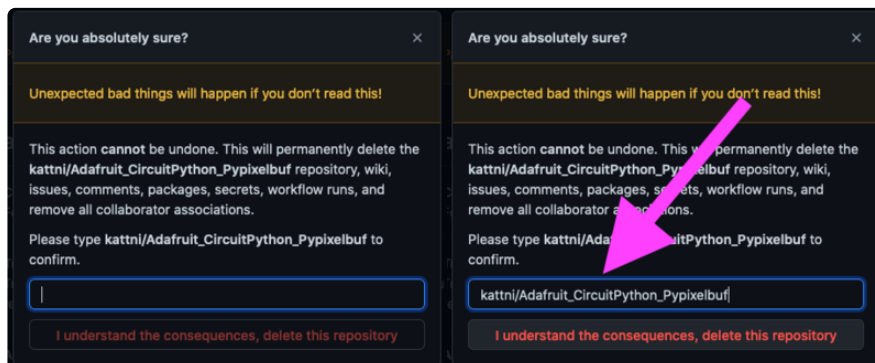


Scroll down on the first page of Settings until you find **Danger Zone**. At the bottom of the Danger Zone, you'll find a button that says Delete this repository. **Click the Delete this repository button.**

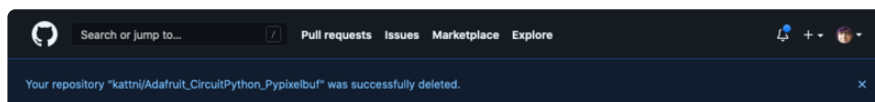


Remember, this will remove the remote copy of your repo. If you have ANY work that you committed and pushed, but did NOT pull request to the original project, IT WILL BE LOST.

Clicking the button will bring up a dialog box. You cannot click the button at the bottom until you enter the full name of the repository, including your GitHub user ID. Enter the full name of the repository including your GitHub user ID, and click the "I understand the consequences, delete this repository" button.



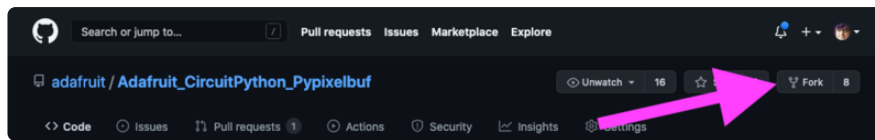
You may be asked to verify your two-factor authentication (if enabled) or enter your password to continue. This will take you back to your main GitHub page including a banner indicating you have successfully deleted the repo.



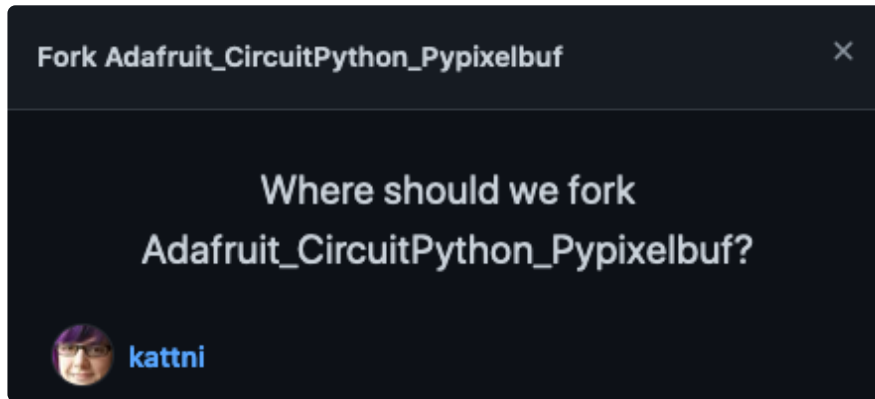
Now that you've deleted your fork, it's time to fork again.

Forking Again

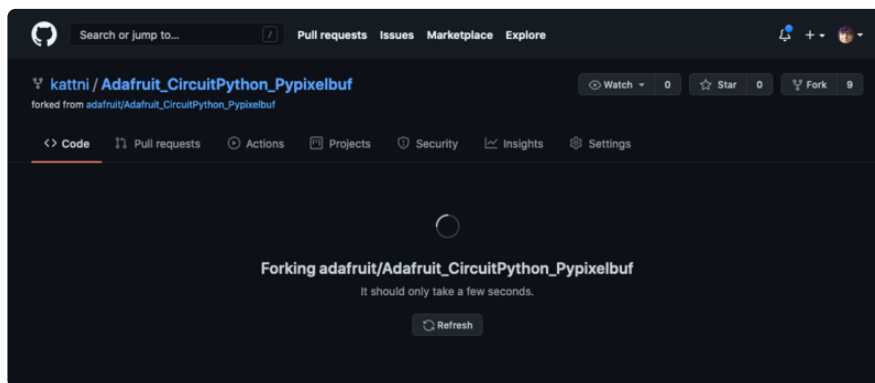
Now, you'll want to fork the repo again. Navigate to the repo in your browser. This time, click the **Fork** button.



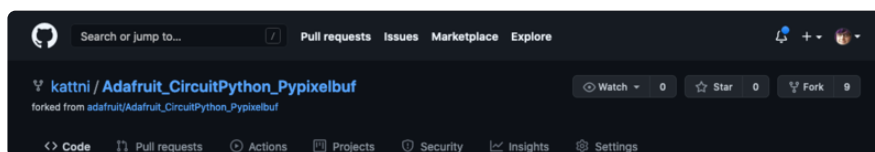
This will bring up a dialog box asking "Where should we fork repo_name?" where repo_name is the name of the repository you're forking. **Click on your GitHub user ID.**



GitHub will begin the process of forking the repository. This can take a few seconds.



You will then be directed to the main page of your new fork.



Now it's time to create a new local clone.

Cloning Again

Now that you've forked the repository again, it's time to download a copy of the repo to your computer so you can begin working with it again. I will quickly go over the cloning process. For a detailed explanation, please see the [Clone Your Repo section of the Grab Your Fork page \(https://adafru.it/JCR\)](https://adafru.it/JCR) in this guide.

First, get the URL. Click the green Code button on your fork of the repo on GitHub, and copy the .git URL, e.g. `https://github.com/kattni/Adafruit_CircuitPython_Pypixelbuf.git`.

Next, from within the directory where you store your Git repos, run the following command:

```
git clone -o youruserid https://your-fork-URL
```

```
3645 kattni@robocrepe:repos $ git clone -o kattni https://github.com/kattni/Adafruit_CircuitPython_Pypixelbuf.git
Cloning into 'Adafruit_CircuitPython_Pypixelbuf'...
remote: Enumerating objects: 343, done.
remote: Counting objects: 100% (170/170), done.
remote: Compressing objects: 100% (121/121), done.
remote: Total 343 (delta 73), reused 121 (delta 43), pack-reused 173
Receiving objects: 100% (343/343), 101.59 KiB | 1.69 MiB/s, done.
Resolving deltas: 100% (166/166), done.
```

Navigate to the newly created directory by using `cd directory-name`.

```
3646 kattni@robocrepe:repos $ cd Adafruit_CircuitPython_Pypixelbuf/
3647 kattni@robocrepe:Adafruit_CircuitPython_Pypixelbuf $
```

Finally, include the original project as a remote by running the following, replacing `ownerID` with the original project owner's GitHub ID, and the URL with the original project's .git URL. You can obtain the original project's .git URL by clicking the green Code button on the original project's GitHub repository, and copying the URL.

```
git remote add ownerID https://original-project-url
```

```
3647 kattni@robocrepe:Adafruit_CircuitPython_Pypixelbuf $ git remote add
adafruit https://github.com/adafruit/Adafruit_CircuitPython_Pypixelbuf.git
3648 kattni@robocrepe:Adafruit_CircuitPython_Pypixelbuf $
```

Now you're entirely up to date again and ready to begin working with the code!

For details on the next steps, check out the rest of this guide.

Glossary

This page includes a list of terms used in this guide with definitions.



add:

`add` is the command used to stage a changed file for commit. When you add a file, it changes the `status` from `Changes not staged for`

`commit:` to `Changes to be committed:`. This means when you next commit, any files you `add` will be included.



branch:

A branch is a way to have your own working timeline of changes. Creating a working branch of your own is a way to make changes while leaving the default main branch clean. You can always merge your working branch changes into main at any time.



cd:

`cd` is the command to **change directory** from the command line. You'll use this to navigate through your local repos on the command line.



change request:

A change request is a request for changes as part of a review on an open PR.



checkout:

`checkout` is the command used to switch to a new branch, by creating it in the process, or to switch to an existing branch. Using it with `-b` will create a new branch. Using it alone will switch to an existing branch.



clone:

Cloning a repository creates a local copy of the repo on your computer. It is good practice to use `git clone -o alias repo-url` to assign your own alias to the upstream remote to avoid confusion. Simply cloning a repo using `git clone repo-url`, uses `origin` for the name of the upstream remote.



commit:

A commit is a save point in your project. It's similar to saving a file to your computer, however, instead of overwriting the previous save, it creates a timeline of save points. You can return to a previous save point at any time. The `commit` command creates a commit. It is most easily used with `-m "Commit message"` to include your commit message.

You can use committing often to divide up your set of changes. Consider a commit to be a complete and distinct idea. Each time you complete a concept you wanted to change, commit. The sum of these commits will be a combination of all the changes you intend to submit to the final project. This creates a timeline for your set of changes and allows for a better understanding of what your train of thought was while you were completing them. This can make it easier for you to make changes later, and easier for a reviewer to see where you were going with your ideas.



continuous integration testing:

Continuous integration testing allows for automatically checking code that is submitted to a repo for style and syntax errors, among other things, to verify that the code is ready to be merged. It ensures that the submitted code will build successfully, without requiring someone to go through each contribution to try to find the errors manually.



diff:

A diff is the difference between two files, sets of changes, or commits. When you run `diff`, it shows you the changes you've made since your last commit, or since you opened the original file if you have not yet made any commits. It provides a color coded look at the difference between the two states, which highlights all the changes you've made. It only shows you the code near your changes - some files are extremely large and it would take forever to scroll through the entire file to look at a small change. Be aware, there are times when you'll make many changes, and the results of `diff` will take a long time to go through.

When you view the diff as part of a pull request, it shows you all the changes included in that PR. It also only shows you the code around the changes to conserve space.



fetch:

Fetching is the act of grabbing the changes from a remote repo, but not merging them in. You'll use `fetch` when you're preparing to update your main branch to be in line with the original project.



fork:

A fork is a copy of the original project that lives on your GitHub account. You clone your fork locally and it allows you to work on the project without affecting the original. Forks remain attached to the original project which allows you to submit pull requests with changes you'd like to see merged into the original project. You can also keep your fork updated by fetching updates from the original project repo.



Git and git:

Git is the actual free and open source distributed version control system that you're using locally to work with your repo. `git` is the beginning of every Git command, such as, `git commit` or `git checkout`.



linting:

Linting is the process of checking code for style and syntax errors. A linter is the tool used for linting. When Travis CI runs on your pull request to an Adafruit repo, it's running a linter called Pylint on your code to verify that it is in line with Adafruit's required standard.



main:

The default branch is called main. It's good practice to make changes on a working branch and leave the main branch clean.



merge:

A merge takes the changes from one place and merges them into another. Your changes will be merged following an approved pull request. You'll `merge` after you `fetch` the changes from a remote repo to update your main branch.



pull request or PR:

A pull request or PR is a request for your changes to be merged with the original project. Consider a PR to be a conversation. Some PRs will be accepted immediately, however, most will involve some form of discussion or change request. A PR is not a single step, it is a process. You'll create your PR, submit any fixes necessary for the checks to pass, wait for review, submit any or discuss changes requested in the review, and then wait for your code to be merged into the project. Not all PRs will be accepted. This is why it's important to submit a PR earlier rather than later so you can get feedback earlier on in the development process.



push:

`push` is the command used to send the list of commits since the last push to your remote repo. In other words, you're "uploading" your changes to your repo on GitHub. Until you `push`, none of your commits show up on GitHub. So think of commits as local save points, and pushes as remote save points. This also means that once you `push`, your changes are visible to the public. So `commit` as often as you like, but only `push` when you're ready for it to be submitted to the project. If you do `push` too soon, it's okay though! It happens to all of us. You can always `push` again after you do a few more commits.



remote:

A remote is the version of a repo located on GitHub. You work on the repo locally and then push your changes to your remote. The `remote` command allows you to create aliases to your remote repo and the original project remote repo for the purposes of pushing changes and keeping your repo and fork up to date.



repository or repo:

A repository can be thought of as a project folder. It includes all the files contained within the project. Use GitHub to create your own copy of a project you'd like to contribute to. Then use Git to download your repo to your computer so you can make your changes locally.



review:

A review is the process of someone going through a pull request to verify that it's done correctly, and to decide whether it's appropriate to merge into the original project. Some reviews are quick, requiring only that the code be verified. Others will take a significant amount of time, involving an extensive conversation with change requests and suggestions for improvements. Reviews are meant to be a positive experience for everyone involved, and ensuring that any feedback provided is positive and constructive is an essential part. Anyone is welcome to provide a review on a pull request, as long as they provide constructive, positive feedback.



staged:

When you've made changes but have not included them for commit, they are considered to be not staged for commit. When you have run `add` to include your file for commit, your changes are considered to be staged to commit. When changes are staged for commit, this means they will be included in your next commit.



status:

`status` is the command that shows you the current status of your changes. You should run `status` before running every other command you intend to run. While it's unnecessary with some commands, using it consistently will get you in the habit so you never miss it when you do need it. When you run `status`, you'll not only find out the current status, you'll know what command you need to run next based on the current status. `status` is your best friend!



Travis CI:

Travis CI is the continuous integration testing system built into Adafruit repos to verify that all submitted code builds successfully, and to check code for style and syntax errors. This is the system that will tell you if your code fails the check, and then provide you with a log showing you a detailed list of the errors.



upstream:

You forked an original project and then cloned that fork locally. The original project is often referred to as upstream from your fork.