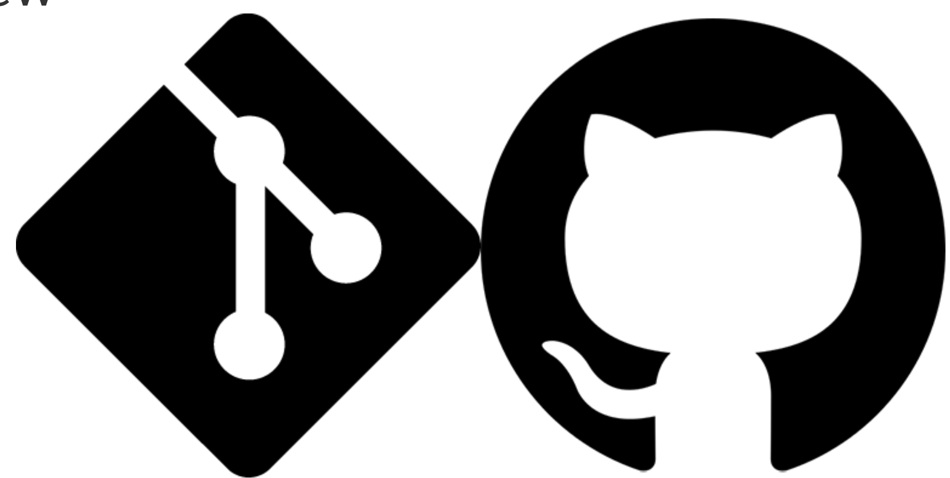
Contribute to CircuitPython with Git and GitHub

Created by Kattni Rembor

Last updated on 2020-12-09 02:15:58 PM EST

Overview



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) 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/BI0). 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) that you must complete. You must have Git installed and setup on your computer (https://adafru.it/BI3). You must have a GitHub (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). Further information is likely available through the Git documentation (https://adafru.it/BI1) or the GitHub documentation (https://adafru.it/BI2).

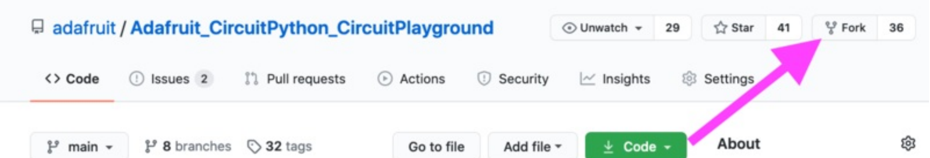
Though I primarily work with CircuitPython and related projects, the workflow outlined in this guide should apply to any open source project. However, it's always a good idea to check with the project's maintainers to make sure that you're working within their guidelines.

Let's get started!

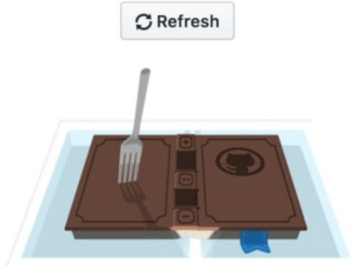
Grab Your Fork

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 Circuit Playground Express 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.

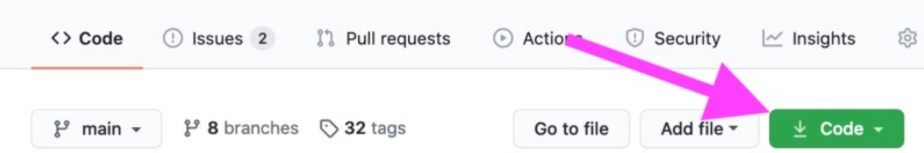


Great job! You now have your very own GitHub copy of the project repo to which you're going to contribute. You're ready for the next step!

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**.

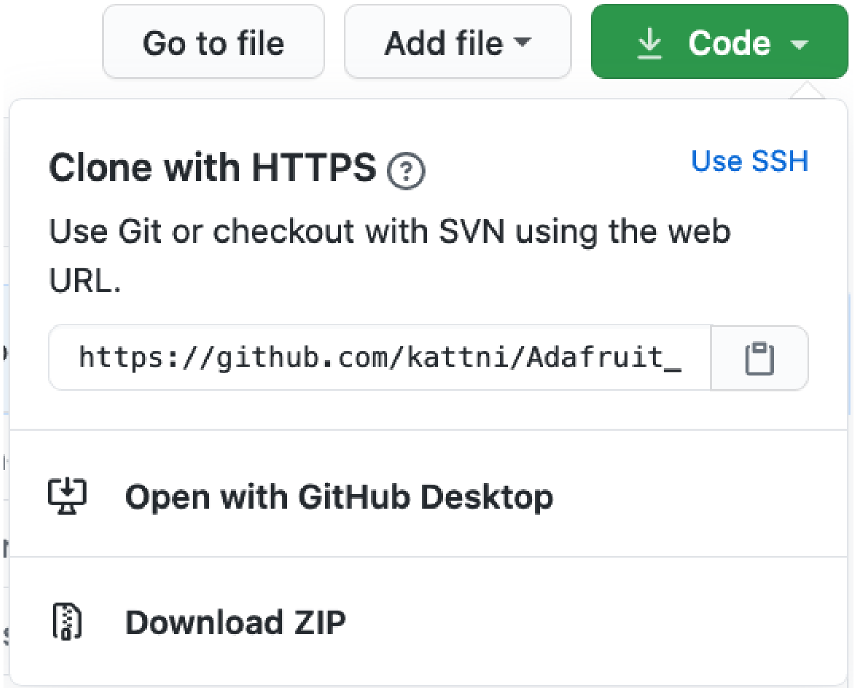
On the page for your repo, you'll find a green **Code** button.



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

https://github.com/kattni/Adafruit_CircuitPython_CircuitPlayground.git **(https://adafru.it/BHX)**

Click the **Copy** button to copy the URL to your clipboard.



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

```
1430 kattni@robocrepe:~ $ 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 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 `youruserid` with your GitHub user ID, and paste the URL from your clipboard:

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

```
1431 kattni@robocrepe:repos $ git clone -o kattni https://github.com/kattni/Adafruit_CircuitPython_CircuitPlayground.git
Cloning into 'Adafruit_CircuitPython_CircuitPlayground'...
remote: Counting objects: 330, done.
remote: Compressing objects: 100% (37/37), done.
remote: Total 330 (delta 22), reused 44 (delta 14), pack-reused 276
Receiving objects: 100% (330/330), 10.05 MiB | 3.61 MiB/s, done.
Resolving deltas: 100% (144/144), 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_CircuitPlayground**, which contains a copy of the newly forked repo.

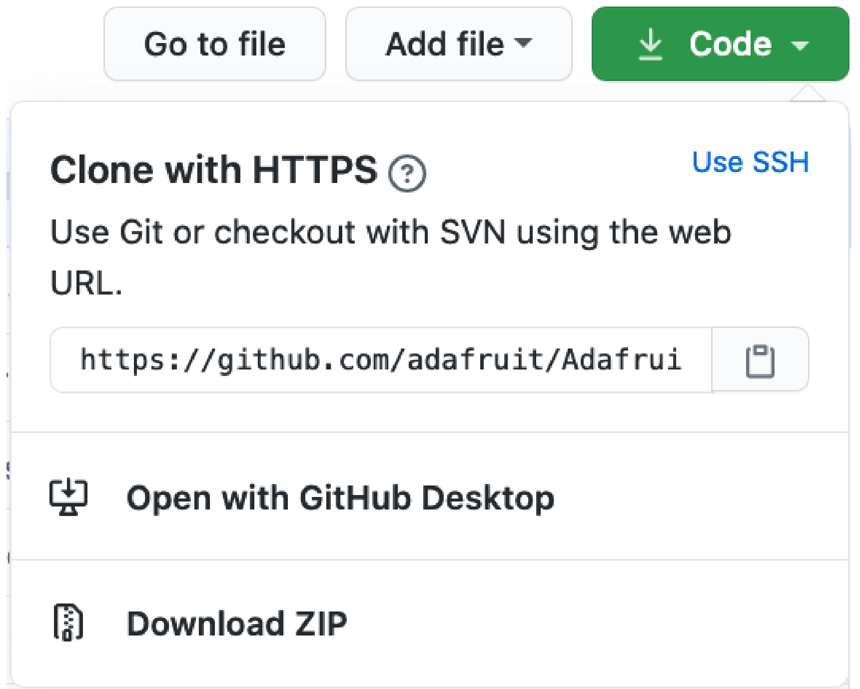
Now, use the `cd` command to move into that directory.

```
kattni@robocrepe:~ $ cd repos/Adafruit_CircuitPython_CircuitPlayground/
```

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 `ownerID` to the original project owner's GitHub ID, and pasting the URL from the clipboard:

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

```
$ git remote add adafruit https://github.com/adafruit/Adafruit_CircuitPython_CircuitPlayground.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

 We are currently in the process of moving away from master being the default branch, to main being the new default branch. This page still refers to both, but expect that it will eventually refer only to main as the default branch. Please read this page thoroughly as it will soon include instructions for updating your local clone from master to main as necessary.

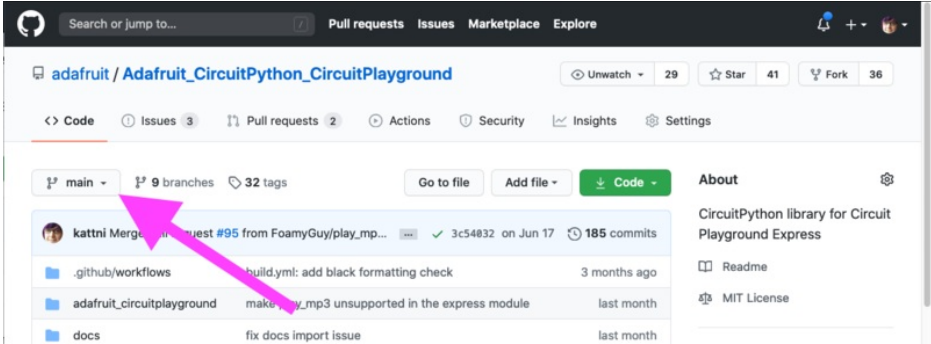
Now that your local repo is set up and ready to go, it's time to start working with it.

Starting from the Right Place

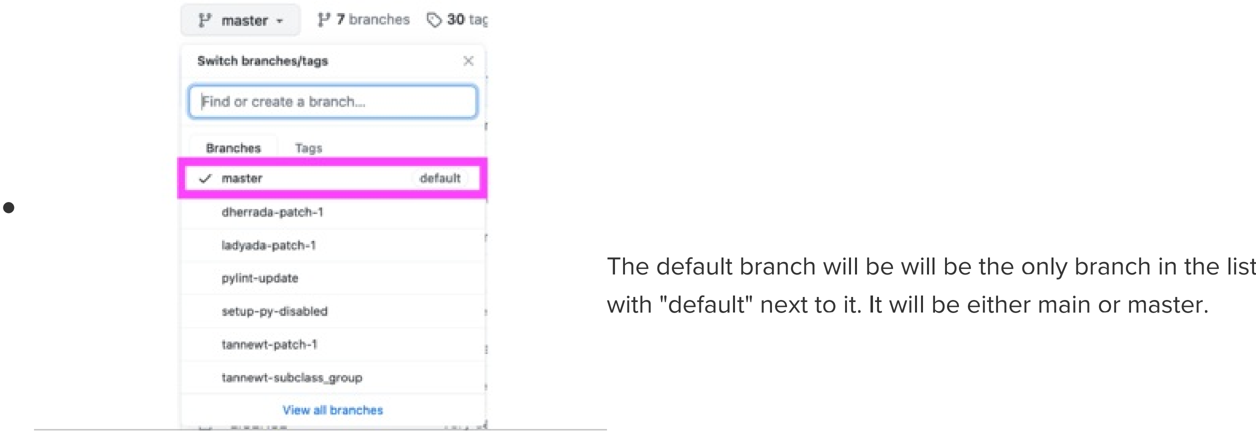
Imagine you made a change to your code, but made a mistake. Now your repo is in a bad state. To help avoid this situation, we use branches. You always want to make changes while on a **branch**. A **branch** is a way to have your own working timeline of changes, while leaving the default branch even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

Main or Master?

The first thing you need to do is determine is whether the library you are working with is using **main** or **master** 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. It will be either main or master.

Updating the Main or Master 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 or master 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 or master, first checkout main or master 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'.
```

Or:

`git checkout master`

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

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
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  master     -> adafruit/master
```

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 or master from the remote repo and combine it with the main or master branch on your local repo. This will bring you even with the remote main or master, and that means you're up to date.

To merge the remote main or master with your main or master, run the following `merge` command, replacing `ownerid` with the name you assigned to the original project's 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(-)
```

Or:

`git merge ownerid/master`

```
1488 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git merge adafruit/master
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 or master since I last did anything with this repo. Good thing I updated!

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

Alternatively, you can simply run `git checkout ownerid/main` or `git checkout ownerid/master` (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 or master 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 or master branch, you can enter:

`git checkout main`

or

`git checkout master`

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

Moving your Local Clone and Fork from master to main

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

 If you'd rather avoid these steps, AND you have NO UNCOMMITTED WORK IN PROGRESS, you can alternatively delete your clone from your computer and your fork from GitHub, and then follow the steps in the guide again to refork and reclone the repo. DO NOT DO THIS IF YOU HAVE UNCOMMITTED WORK IN PROGRESS, YOU WILL LOSE YOUR WORK.

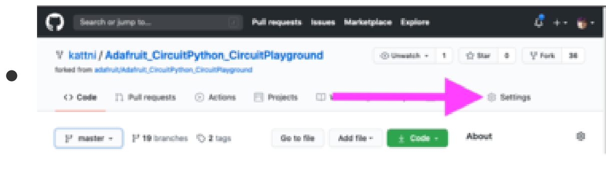
Once the original Adafruit GitHub repository has moved to main, you will want to update your fork to have main as the default branch as well. Run the following commands from command line (in your terminal window):

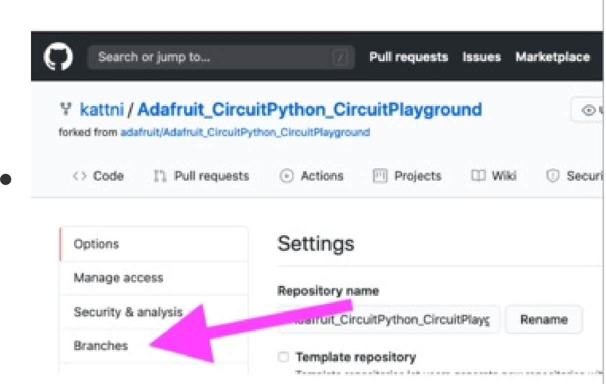
```
git checkout master
git pull
git checkout main
git push YourUserID main
```

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:

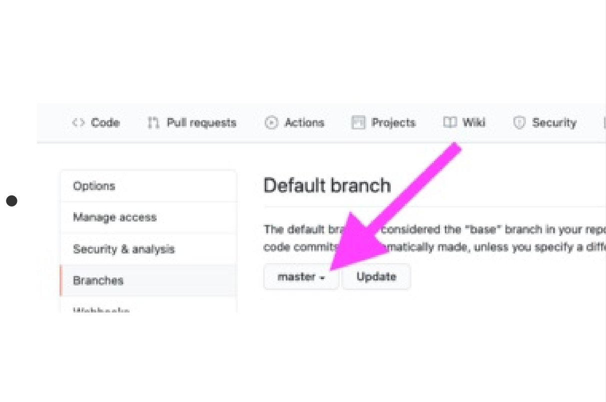
https://github.com/kattni/Adafruit_CircuitPython_CircuitPlayground (https://adafru.it/MeN)

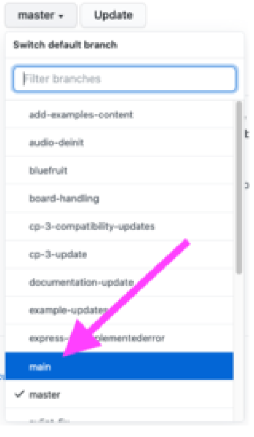
Once there, complete the following steps:

- 

- 

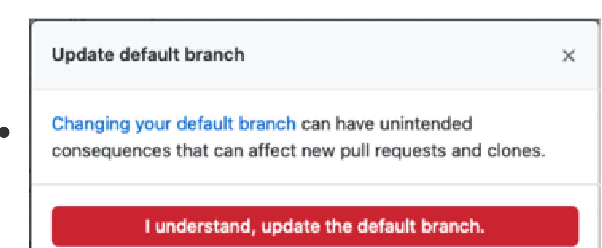
- Click **Settings**, the right-most tab below the repo name.
- Once in Settings, click **Branches** on the left.

- 

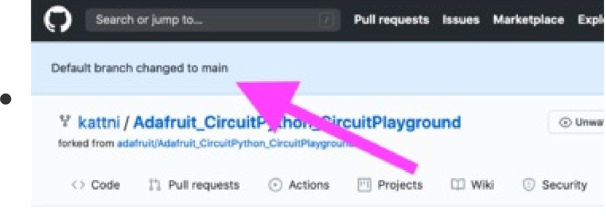
- 

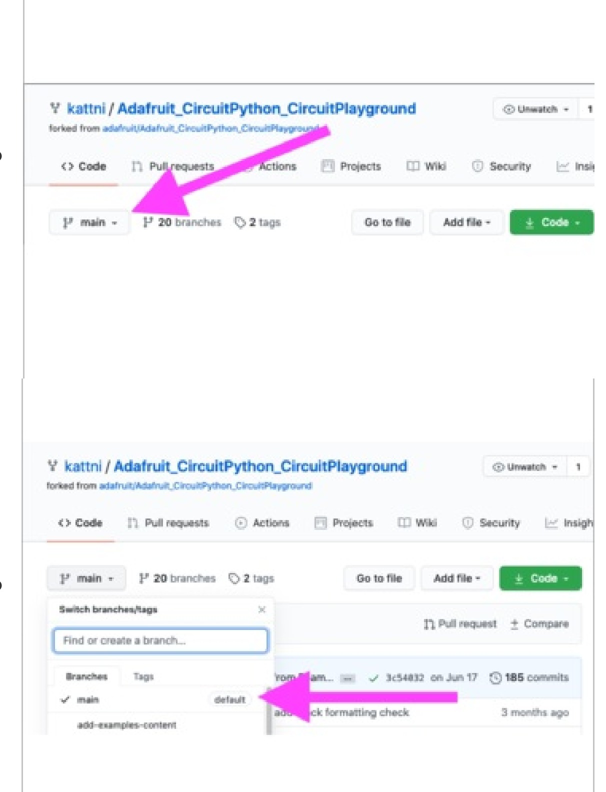
- Default branch

  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.

  main
  Update
- 

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

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.

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). Typically it defaults to vim. Windows users will probably want to check here (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`

```
1470 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git push kattni play-file-stop-tone-fix
Counting objects: 12, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (9/9), done.
Writing objects: 100% (12/12), 1.14 KiB | 390.00 KiB/s, done.
Total 12 (delta 6), reused 0 (delta 0)
remote: Resolving deltas: 100% (6/6), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_CircuitPlayground.git
 * [new branch]      play-file-stop-tone-fix -> play-file-stop-tone-fix

1471 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

Depending on your Git setup, you may be asked to provide login information when you push. In this case, you'll provide your GitHub credentials.

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

Not `push`ing to the CircuitPython `master` Branch?

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).

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.

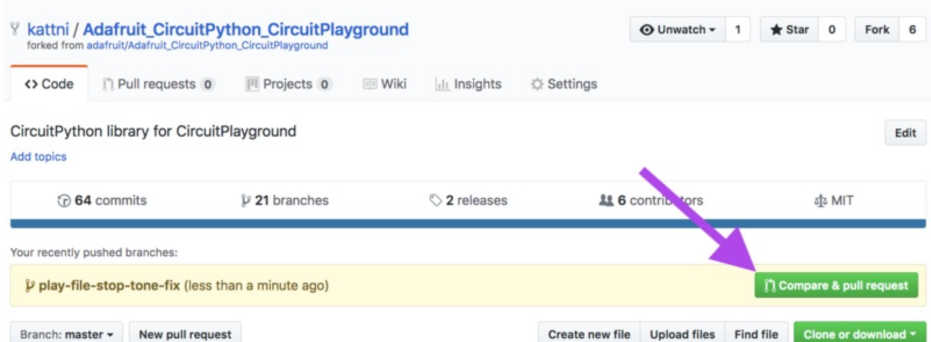
A pull request isn't a single step, however. It's a process. You'll create your PR, submit any fixes necessary for the checks to pass, wait for review, submit any or discuss changes requested in the review, and then wait for your code to be merged into the project. Not all PRs will be accepted. This 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

Once you've pushed your changes to your fork, and you're ready to submit them to the project, open your browser and navigate to your forked repo.

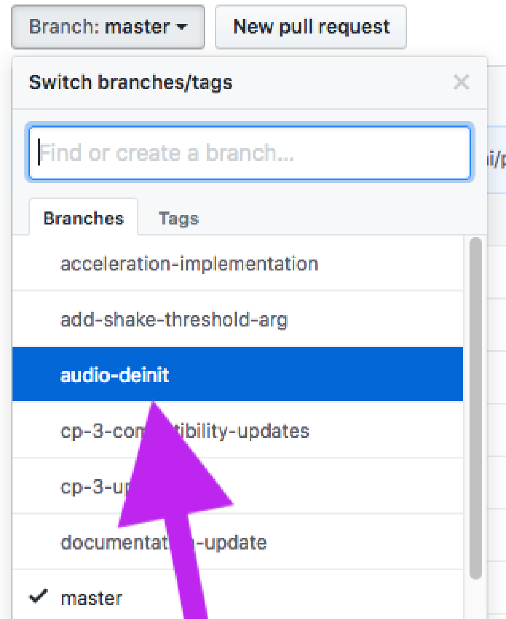
If you've just pushed for the first time, you'll see a line at the top stating, "You've recently pushed branches:" and a bar below it containing your branch name and a **Compare & pull request** button.



If you pushed multiple times from the same branch in a short period of time, or for some other reason the Compare & pull request button doesn't show up, you can create it manually.

Click on the **dropdown menu for Branch:**

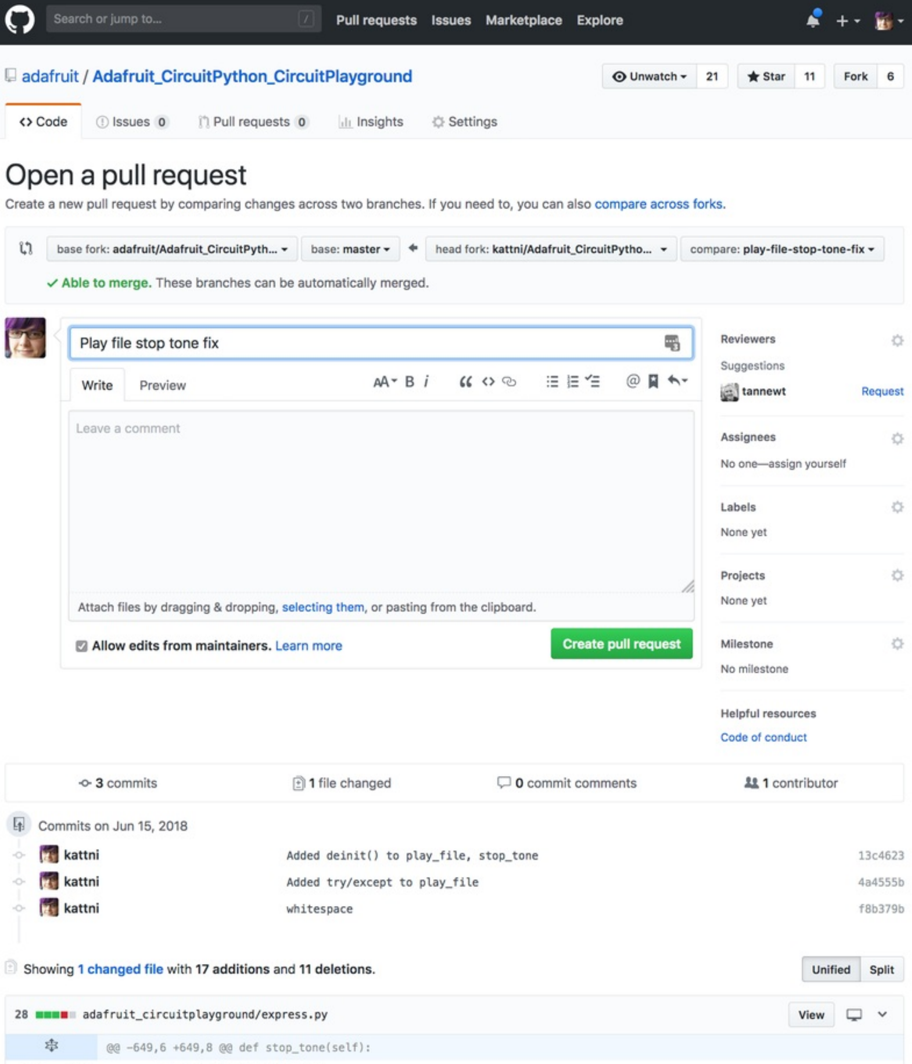
Find **your branch name** in the menu.



Then click the **New pull request** button.

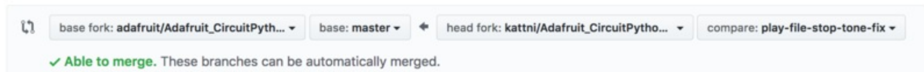
Open a Pull Request

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



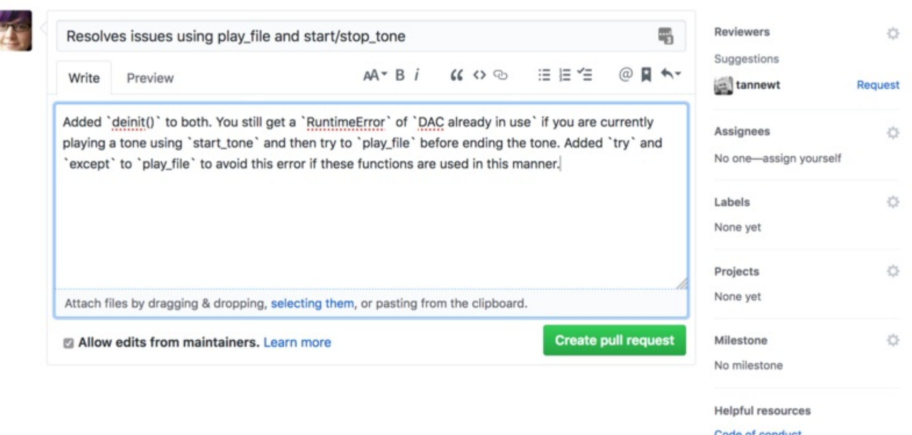
Let's break it down!

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.**

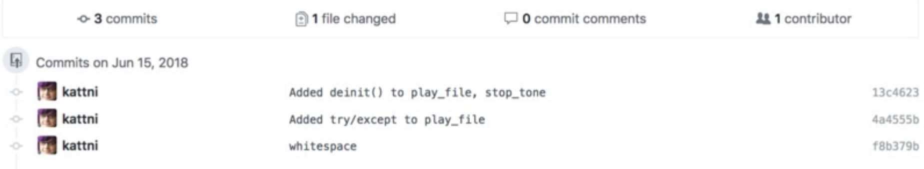


The next section is where you'll enter your pull request message. Ideally you'll title it something that quickly describes what you changed. Then you can include more details in the message body. If you made a single commit, they may already be populated by your commit message. If you made multiple commits, 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.

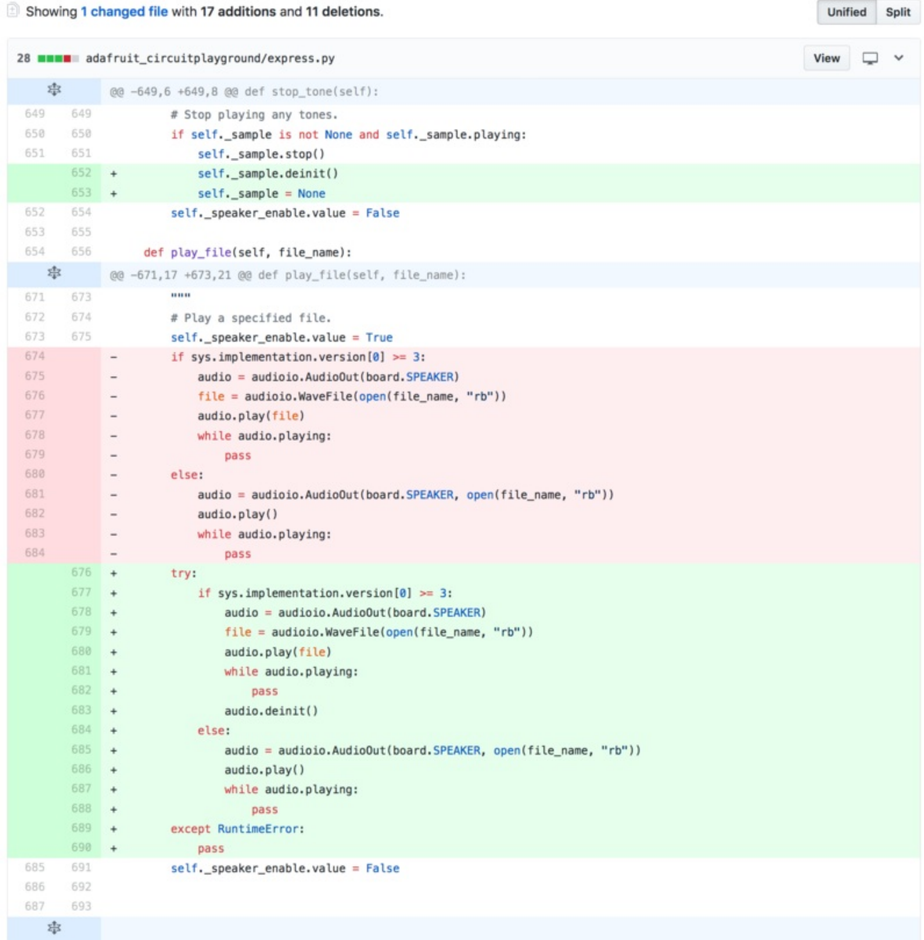


The next section includes your list of commits. The top details the number of commits, how many files were changed, the number of commit comments, and the number of contributors. I made 3 commits, changed one file, have no comments and I'm the sole contributor.



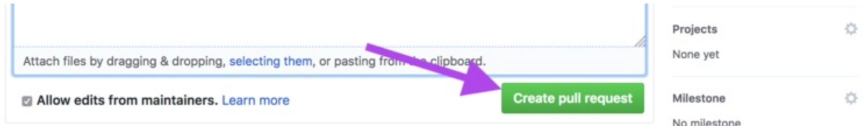
The last section shows you your changes. Like `git diff`, it will show you only the code surrounding your changes. The code you've added will show up in green. Any code you deleted will show up in red. Be aware, if you change an entire code block, it will show the original code in red, and your new code 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'll see if you look at the changes I made that I added a try and except around an existing code block. It shows the original code block in red. The new code block in it's entirety is green, even though I really only added three lines of code.



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, you'll find the **Create a pull request** button. Click it to create your pull request.



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

Open Pull Request

Now you've created your pull request and you're ready to continue the process. In this section, we're going to cover how to deal with the built in checks failing, and how to submit code to an active pull request. You'll learn to read the logs generated by the check system, and how to take that information and apply it to fixing your code.

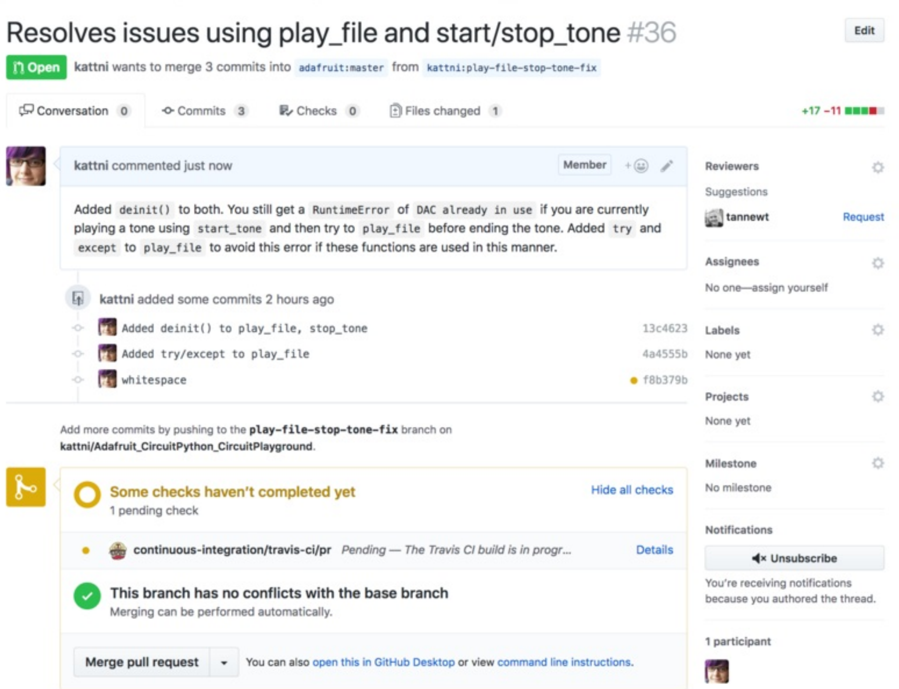
Many of the Adafruit repos have a built in checks system called **Travis CI**, which does what is called **continuous integration testing** (the CI in Travis CI). This check will verify your code for style and syntax, and make sure that the everything builds with your code included.

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 Travis. It will contain any errors found and allow you to more easily find them in your code.

Once you've clicked the **Create pull request** button, it will automatically take you to your newly created PR. Let's take a look.

Pull Request Explored

The title of the PR will be what you entered into the message title when you created it, followed by the PR number.



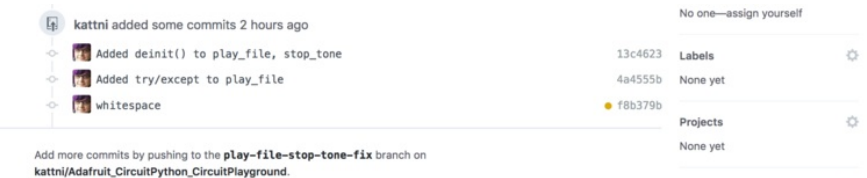
The next section tells you the current status of the PR. This PR is **Open**. I am currently **requesting to merge 3 commits into `adafruit:master` from `kattni:play-file-stop-tone-fix`**. Remember, I'm asking to take the code from my branch and add it to the master branch of the original repo.

Open kattni wants to merge 3 commits into `adafruit:master` from `kattni:play-file-stop-tone-fix`

The next section includes the body of the message you submitted with your pull request. The tabs at the top lead you to different sections of the PR. You can view a list of the commits, the checks, and the files changed in a diff format like included when you created your PR. Click on each one to view its contents.

The upper right of this section includes some green and red numbers and boxes. These indicate the number of lines of code added and removed. You'll find they reflect the color coding found in the Files Changed section.

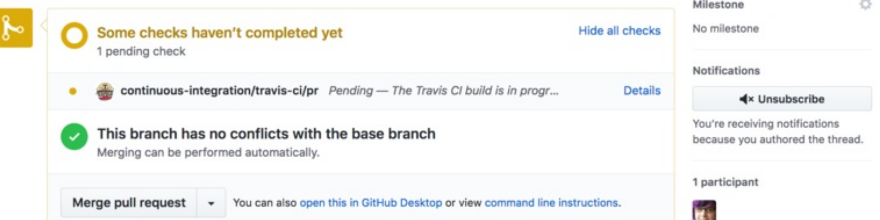
Next is a list of the commits included with the initial PR. You'll notice the last commit has a yellow dot next to it. This is because the checks have not yet completed.



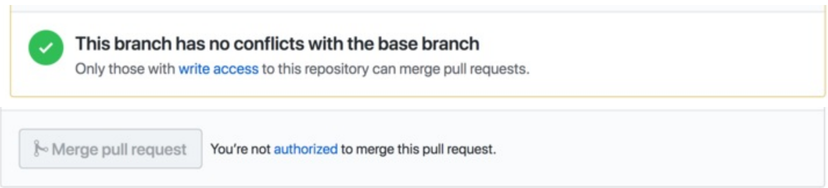
After the PR is created, Travis will begin it's testing. Depending on the size of the repo, this can take anywhere from **a few seconds to several minutes** . Until it's completed, you'll see the final commit has a yellow dot and the beginning of the next section will be yellow and say "Some checks haven't completed yet".

The second part of this section should say "This branch has no conflicts with the base branch". If you created your pull request when it didn't say "Able to merge" in green at the top, then this might look different.

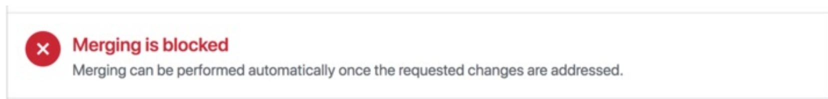
The last part is the merge button, which won't turn green until everything is set. I have merge permissions on this repo, so the Merge pull request button shows up for me.



If you don't have merge permissions on a repo, the end of this section may look like either of the following.

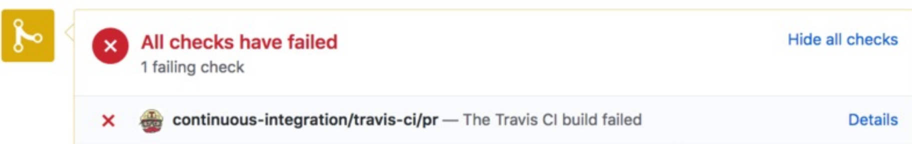


In the event that a repo is setup to require a review before merging is allowed, the last section may include something similar to the following warning in place of the section that says "This branch has no conflicts..."



All Checks Have Failed

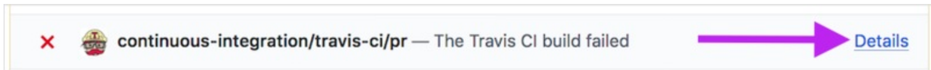
Travis testing completed, and it has failed. This means my PR is not yet ready for review. If your checks fail, you need to work that out before the review process can begin.



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

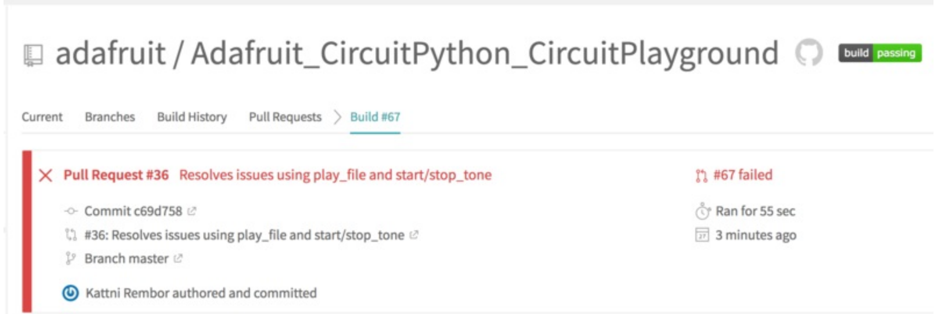
This is going to happen to you. Consider it to be a normal part of the process. We've all dealt with it. The more you go through it, the more you'll learn, and eventually it will happen less. But inevitably you'll miss something, and Travis is here to let you know. Let's take a look at how to get that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

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

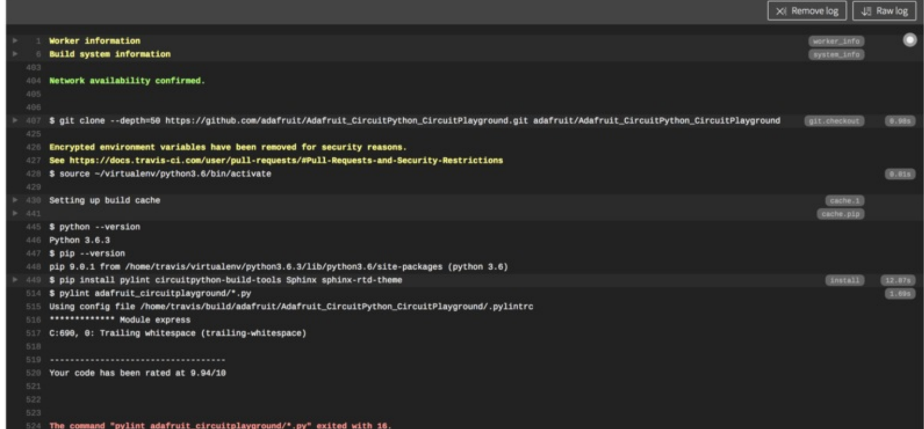


This will take you to the Travis CI log for this check.

The first section outlines the details of the check. It tells you the PR on which it was run, the status of the check, the current commit, how long the check took, when it was run, what branch you're requesting to merge with, and who authored and committed to this PR. The build number is a Travis-specific number.



This check has failed, and I need to know why. The next section is the log generated during Travis' check process. I'm going to scroll down until I see red text, which will indicate the check process exiting on a failure.



Travis has failed during the **linting** process. A **linter** is a tool that checks code for things like errors in syntax or style. Travis uses the **pylint** tool to check code. It is possible to run `pylint` on your computer.

You can run it before you even push to your fork and fix any issues ahead of time. This involves some setup. More information can be found here (https://adafru.it/BI5).

Here, the issue lies in the module `express`. This might seem obvious in this case, since I only edited one file. However, often a PR consists of many files. In that event, the Travis log may contain a list of multiple files with errors in them. Each one will have the errors listed under the file name. Don't be afraid to ask for help with this! Sometimes the errors are not obvious and it's entirely okay to need assistance sorting them out.

In this log, listed under `Module express` is the following line:

`C:690, 0: Trailing whitespace (trailing-whitespace)`

This is telling me that on line 690 in the file express, I have trailing whitespace. Whitespace is any spaces not containing characters in your file. It's super necessary for CircuitPython to work! However, it shouldn't exist at the end of lines or on blank line, so this is considered an error.

The `(trailing-whitespace)` is not Travis or pylint being repetitive. It's the exact nomenclature of the pylint error. This would be used in the event that you chose to deliberately go against the linter, and needed to disable the pylint error in your code.

I checked my file, and sure enough, I have added a space after `pass` on line 690. So, I need to fix that. This means I will be going through all of the steps in the Status, Add, Commit, Push section (https://adafru.it/BHZ) again, starting with deleting the trailing whitespace from my file. Once I've fixed the issue, I'm ready to commit again. First, I'll check the `status` and `diff`.

```
1471 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")

1472 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git diff
diff --git i/adafruit_circuitplayground/express.py w/adafruit_circuitplayground/express.py
index ebd99d9..3ef6548 100755
--- i/adafruit_circuitplayground/express.py
+++ w/adafruit_circuitplayground/express.py
@@ -687,7 +687,7 @@ class Express:     # pylint: disable=too-many-public-methods
                 while audio.playing:
                     pass
         except RuntimeError:
-            pass
+            pass
         self._speaker_enable.value = False


1473 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

I'm satisfied I've made the necessary changes. So, I'm going to `add`, `status`, `commit`, `status`, and `push`

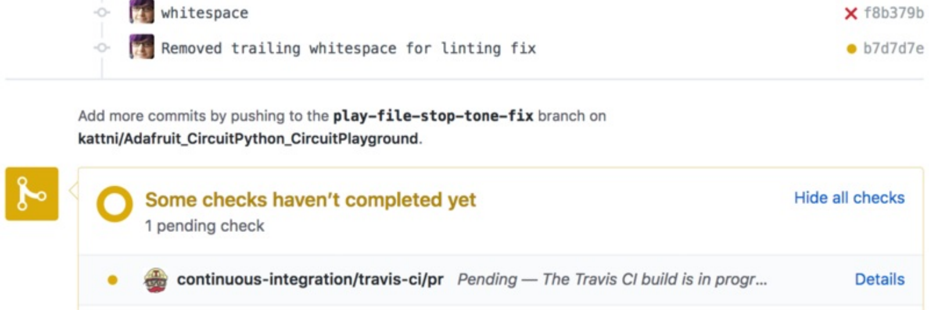
```
1473 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git add adafruit_circuitplayground/express.py

1474 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


1475 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git commit -m "Removed trailing whitespace for linting fix"
[play-file-stop-tone-fix b7d7d7e] Removed trailing whitespace for linting fix
 1 file changed, 1 insertion(+), 1 deletion(-)

1476 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git status
On branch play-file-stop-tone-fix
nothing to commit, working tree clean

1477 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git push kattni play-file-stop-tone-fix
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 369 bytes | 73.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_CircuitPlayground.git
   f8b379b..b7d7d7e  play-file-stop-tone-fix -> play-file-stop-tone-fix

1478 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

 When you push a branch already involved with a pull request, it will automatically update the pull request with your new changes.

You do not need to make a new PR or do anything special to get the current PR to update. All you have to do is push your branch. **This is true for the entire duration of an open pull request, regardless of the reason for submitting updates.**

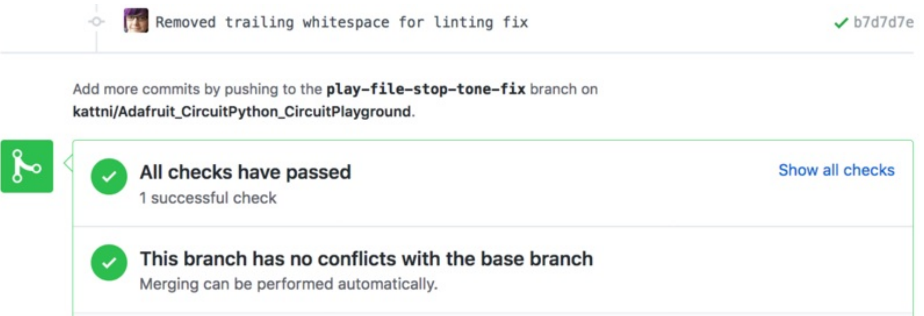
This will trigger Travis to run the check again. The status will return to **Some checks haven't completed yet** during this time.

Note that the previous commit still shows a red X. This is because it failed. GitHub works by tracking a timeline, and that timeline involves a failure. So this commit will always show that it failed even when the fix is applied through a new commit.



Remember, depending on the size of the repo, this can take anywhere from a few seconds to several minutes.

This time, it passes! The status changes to All checks have passed and there's a green check mark next to my last commit.



My pull request is now ready for someone to review it. The next section will cover the process of receiving a review.

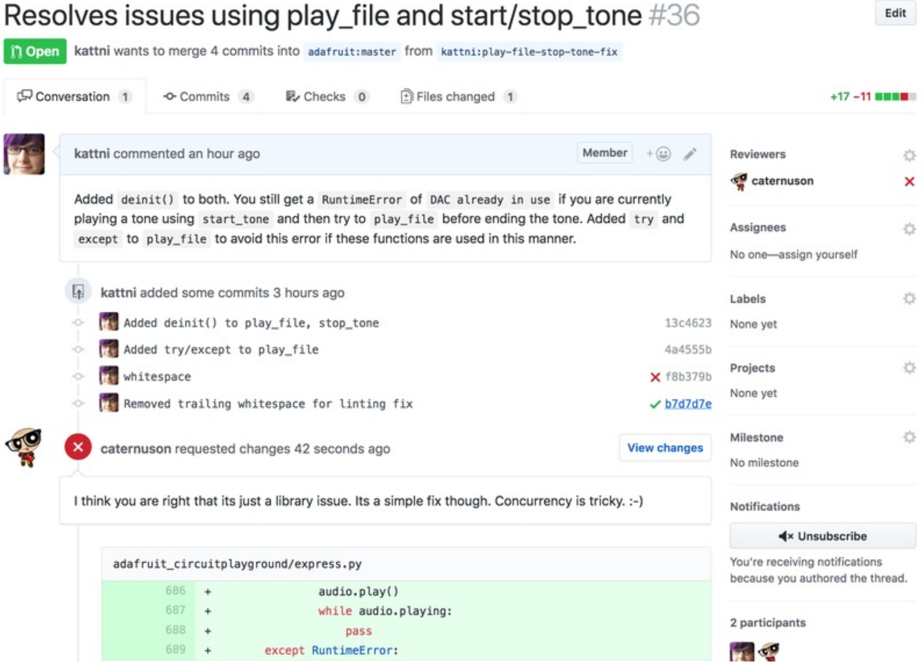
Receiving a Review

The Travis checks on my pull request succeeded and now it's ready for review.

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. But, don't worry, someone will get to it. Your contributions are a huge part of what makes CircuitPython amazing! Providing feedback to help grow our community contributions is incredibly important to us.

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

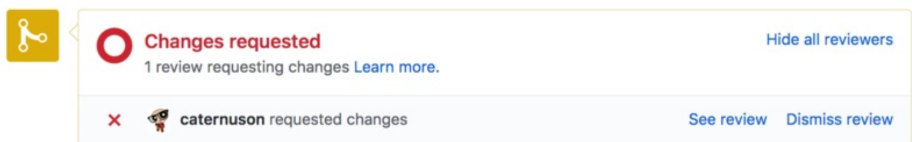


Carter, @caternuson on GitHub, has reviewed my PR! Thanks, Carter!

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

However, this may not always happen. Don't be discouraged if someone requests changes on your PR. It happens all the time. There are many reasons to request changes. The reviewer may have a different perspective on things than you do and suggest a way to do things that you might not have considered, or you may not be aware of a particular format or standard that we follow. It's all part of the review process.

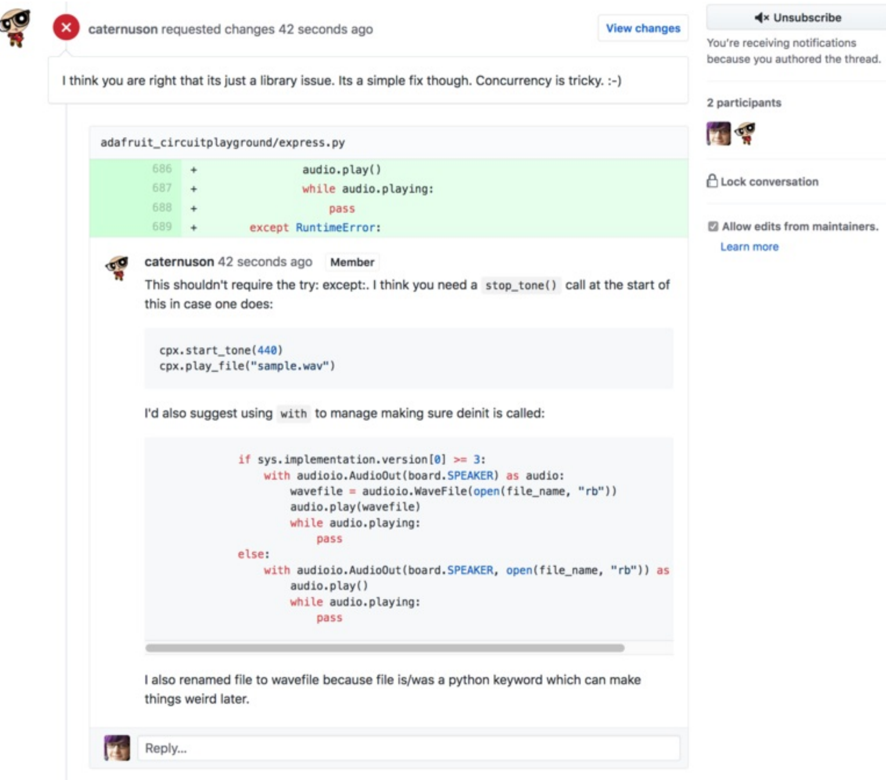
Carter has requested some changes on my PR. The status at the bottom alters to reflect the **Changes requested** status.



As well, there's a red X next to his name under **Reviewers** in the right column, a red circle with an X in it

next to his name above the **change request**. A **change request** is exactly what it sounds like, a request for changes to the code you've submitted for review.

The change request is identifiable because above it, there's a line above it stating **caternuson requested changes 42 seconds ago**. Let's take a look at the requested changes.



With a more involved multi-section review, it's a good idea to click the blue **View Changes** button found on the right side above the change request. This will take you to the Files Changed tab, which would now include a line-by-line review. Since there's only one section to the review, I'm going to view it here.

Carter pointed out that my method for dealing with the `stop_tone` / `play_file` issue could be done in a simpler way. As well, he suggests using a different method to deal with the `play_file` bug I found. These are both great suggestions! I hadn't considered doing it that way, but it definitely makes more sense than the way I did it. So I'm going to incorporate Carter's suggested changes.

Discussing the Review

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** box 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!

Submitting the Requested Changes

Since I agree with Carter's suggestions, I'm going to make the changes. I will again follow the same steps I

did to submit the correction when Travis failed (just as I did in Status, Status, Commit, Push). I begin by adding the changes into my code. I then test it. It works successfully. Great! Time to check the `status` and `diff`.

```
1479 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git diff
diff --git i/adafruit_circuitplayground/express.py w/adafruit_circuitplayground/express.py
index 3ef6548..7330f41 100755
--- i/adafruit_circuitplayground/express.py
+++ w/adafruit_circuitplayground/express.py
@@ -672,22 +672,19 @@ class Express:     # pylint: disable=too-many-public-methods
                   cpx.play_file("rimshot.wav")
         """
         # Play a specified file.
+        self.stop_tone()
         self._speaker_enable.value = True
-        try:
-            if sys.implementation.version[0] >= 3:
-                audio = audioio.AudioOut(board.SPEAKER)
-                file = audioio.WaveFile(open(file_name, "rb"))
-                audio.play(file)
+        if sys.implementation.version[0] >= 3:
+            with audioio.AudioOut(board.SPEAKER) as audio:
+                wavefile = audioio.WaveFile(open(file_name, "rb"))
+                audio.play(wavefile)
                 while audio.playing:
                     pass
-                audio.deinit()
-            else:
-                audio = audioio.AudioOut(board.SPEAKER, open(file_name, "rb"))
+        else:
+            with audioio.AudioOut(board.SPEAKER, open(file_name, "rb")) as audio:
                 audio.play()
                 while audio.playing:
                     pass
-        except RuntimeError:
-            pass
         self._speaker_enable.value = False
```

Those are the changes Carter suggested. Now it's time to check `status`, `add`, `status`, `commit`, `status`, `push`, the same as I did when I was submitting linting fixes.

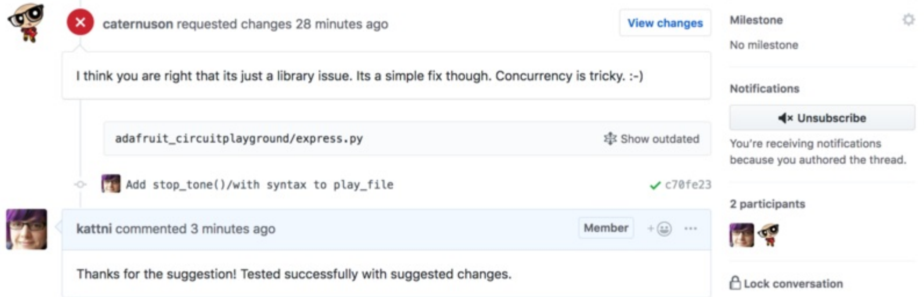
Pushing the current branch will automatically update the open PR following a change request, exactly as it did during the code checking phase. **This is true for the entire duration of an open pull request, regardless of the reason for submitting updates.**

```
1480 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")

1481 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git add adafruit_circuitplayground/express.py

1482 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

1483 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git commit -m "Add stop_tone()/with syntax to play_file"
[play-file-stop-tone-fix c70fe23] Add stop_tone()/with syntax to play_file
 1 file changed, 7 insertions(+), 10 deletions(-)

1484 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git status
On branch play-file-stop-tone-fix
nothing to commit, working tree clean

1485 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git push kattni play-file-stop-tone-fix
Counting objects: 4, done.
Delta compression using up to 4 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 469 bytes | 156.00 KiB/s, done.
Total 4 (delta 2), reused 0 (delta 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To github.com:kattni/Adafruit_CircuitPython_CircuitPlayground.git
   b7d7d7e..c70fe23  play-file-stop-tone-fix -> play-file-stop-tone-fix

1486 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

Now that my changes are submitted, it's time to return to GitHub to view the updated PR.

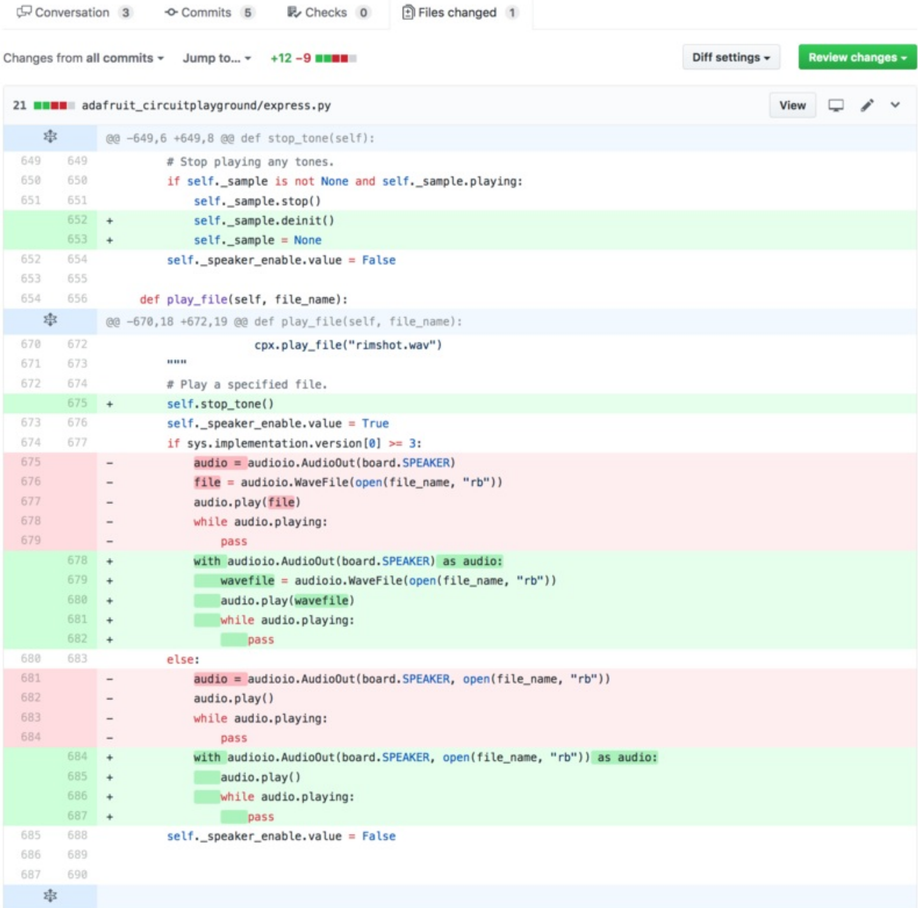
This will trigger a new Travis build. It passed!

Since I addressed all of the requested changes, the change request is now collapsed with the option to **Show outdated**. This is followed by my commit, which has the green check showing it passed the checks. It's always good practice to include a comment to let the reviewer know what you did to address their changes. I've commented to let Carter know that I made the changes and tested them successfully.



The status at the bottom will remain red until Carter approves the changes.

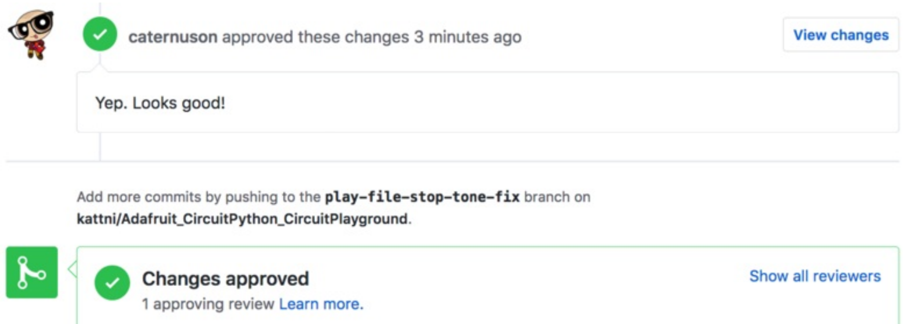
If I click on the Files Changed tab, it will reflect the new changes from the most recent commit. You'll see that the green and red numbers and boxes at the top have changed to reflect the most recent commit as well.



Now I'll wait until Carter has the opportunity to take a look at the updates I made. When he does, I will receive an email letting me know there's been an update to my pull request.

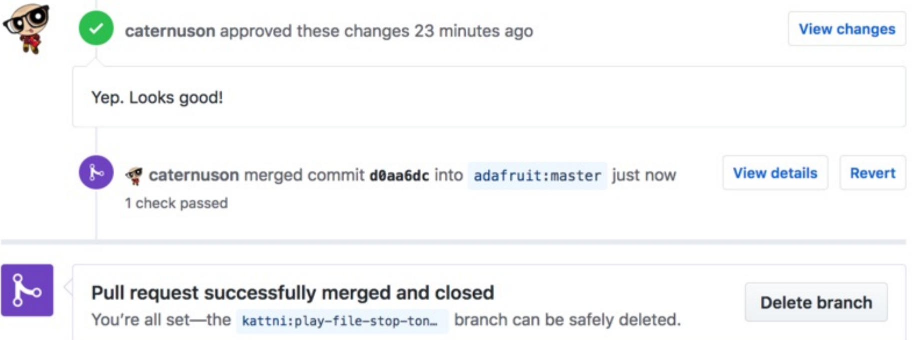
Pull Request Approved

I received the email letting me know there was an update to my pull request. It's been approved! Carter 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.

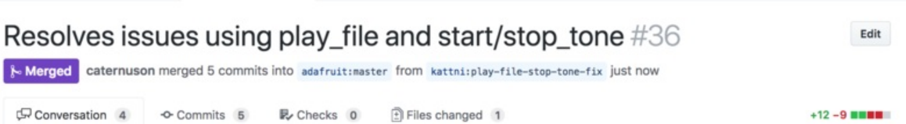


The only thing left is for Carter to merge my changes into the original project. This often happens in quick succession following the approval, and you may never see the status above.

Once the merge is completed, the status changes to **Pull request successfully merged and closed**. The status icons that indicate a successfully merged PR are purple.



The status at the top of the page also changes to purple to reflect **Merged** status. You'll see that it now outlines how many commits were merged, as well as the branches involved.



That's the end of the pull request process. Congratulations! Your code is now merged with the original project and available for everyone to use.

Now it's time to update your own master branch with your new code. The next section shows you how to update your local repo and your fork on GitHub.

Staying Up To Date

 We are currently in the process of moving away from master being the default branch, to main being the new default branch. This page still refers to both, but expect that it will eventually refer only to main as the default branch. For instructions to switch from master to main on your local clone/fork, please see the Always Work on a Branch page: https://learn.adafruit.com/contribute-to-circuitpython-with-git-and-github/always-work-on-a-branch#moving-your-local-clone-and-fork-from-master-to-main-3067724-22

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 or master branch is now behind the original project's main or master branch. It's now time to update your main or master to match the original project.

The first thing you want to do is return to the main or master 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. So let's return to main or master.

To check out the main or master 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'.
```

Or:

`git checkout master`

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

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 or master 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), do e.
From github.com:adafruit/Adafruit_CircuitPython_CircuitPlayground
   862a379..d0aa6dc  master     -> adafruit/master
```

```
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  master     -> adafruit/master
```

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 or master from the remote repo and combine it with the main or master branch on your local repo. This will bring you even with the remote main or master, including the changes you submitted.

To merge the remote main or master with your master, 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(-)
```

Or:

`git merge ownerid/master`

```
1488 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git merge adafruit/master
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 or master 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 or master branch.

Remember, you named your remote repo with your GitHub ID. You'll use this to `push` the updated main or master 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`

```
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 $
```

Or:

`git push yourid master`

```
1489 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $ git push kattni master
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  master -> master

1490 kattni@robocrepe:Adafruit_CircuitPython_CircuitPlayground $
```

Now the main or master 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.

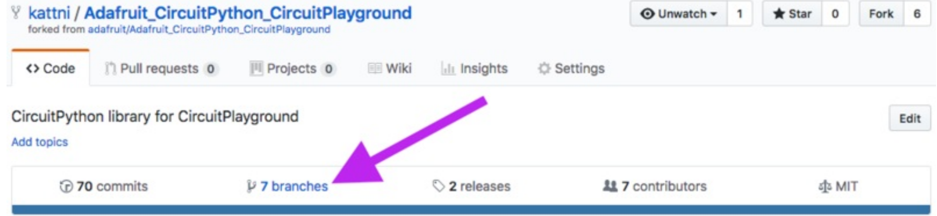
Keep in mind that you have two locations from which branches require deletion: remote and local. Your remote branches are located on GitHub once you push them. Your local branches are located on your computer when you create them.

Deleting Your Remote Branch

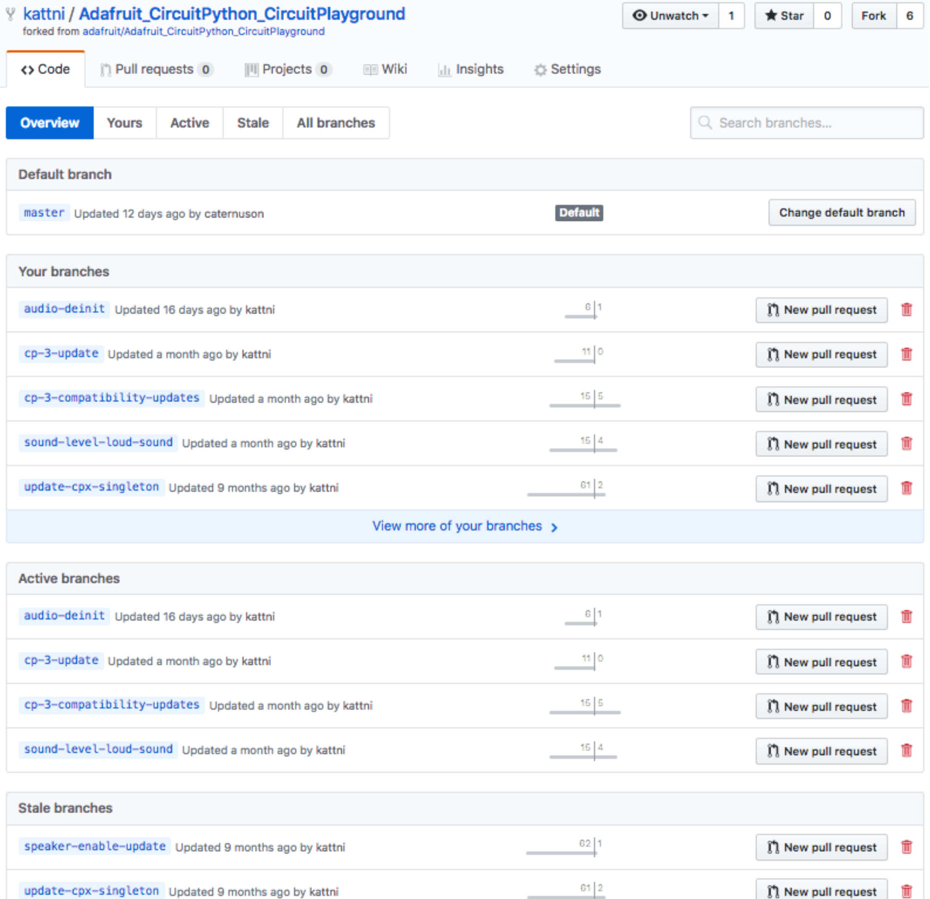
After your PR is merged, there is typically a notification at the bottom that tells you it's now safe to delete your branch, with a button to delete it. If you click this button, it will delete the branch from GitHub, but not from your local copy.

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

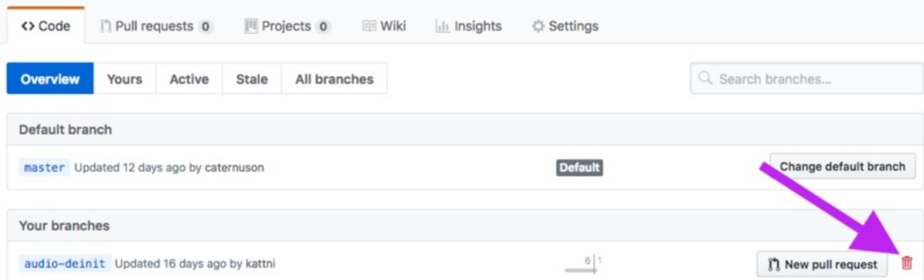
If you're not around when your branch is merged, you can still delete your remote branch from GitHub. Navigate to the main page of your repo, and locate the **# branches** tab, where # is the number of branches you currently have on your repo. In my case, it's **7**. Click the link to the tab.



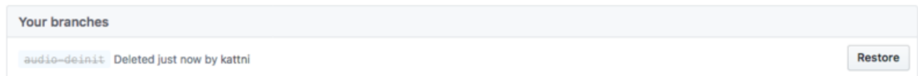
Here you'll find multiple lists of branches. The first is **Your branches**, which contains every branch you have on your repo. The second is **Active branches**, which contains branches that were recently updated. The third, if you haven't kept up with deleting branches as you go, is **Stale branches**, 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 master branch, so first run the following `checkout` command:

`git checkout master`

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

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.

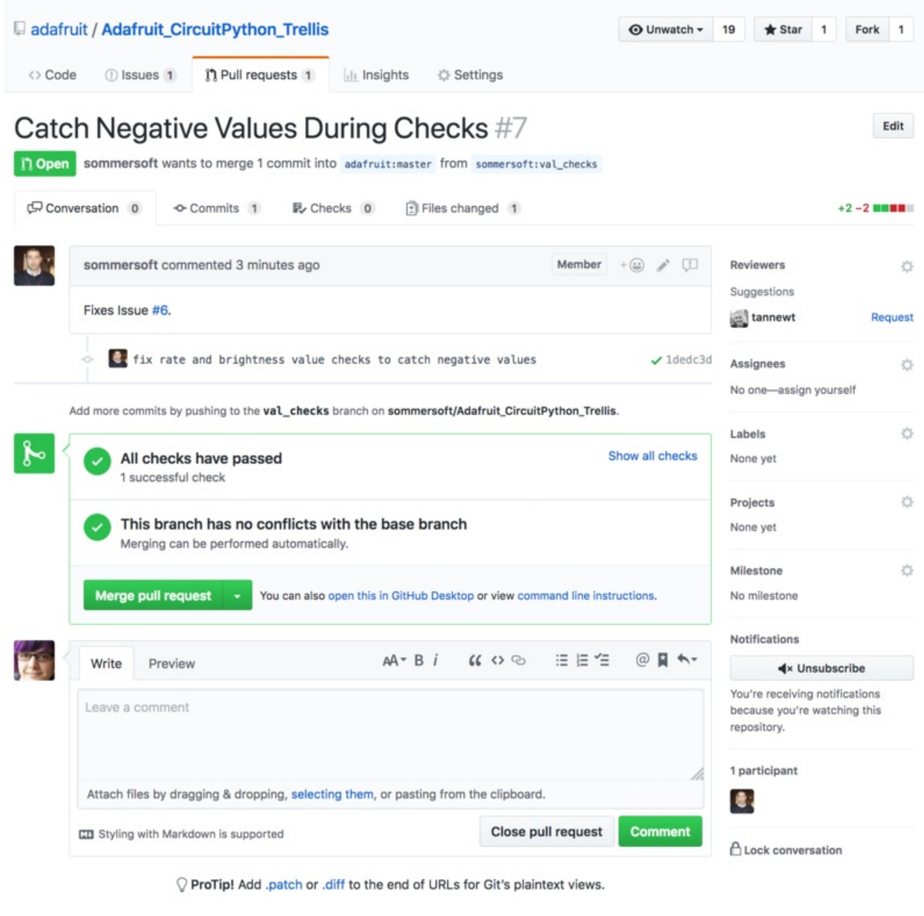
This section walks through the steps of giving a review. In this review, I'll be requesting changes to the code and then verifying those changes were made before approving. Let's get started!

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 Travis to finish testing. If Travis 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 Travis fixes, it's usually a better idea to give the person a chance to take care of the issues first.

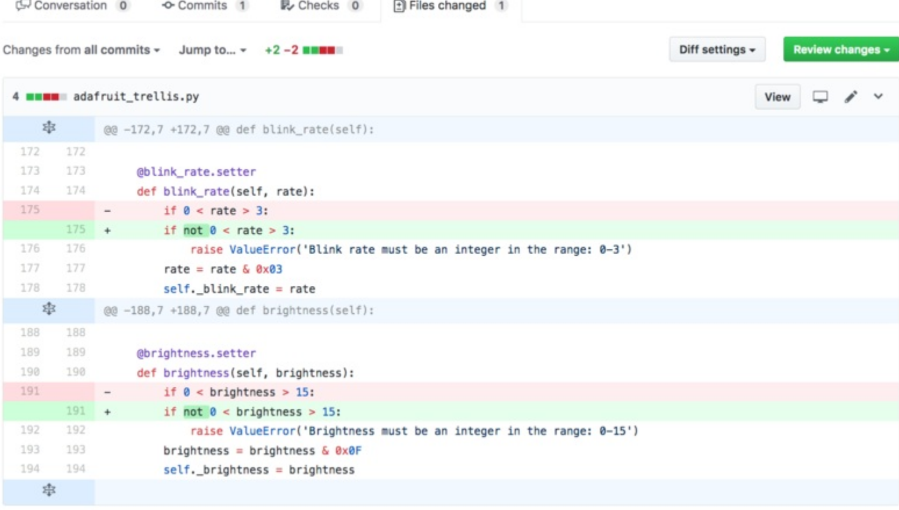
For a detailed look at the elements of a PR, check out the Pull Request Explored section of Open Pull Request.



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.



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.

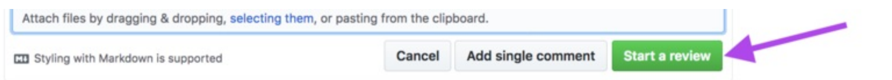
In the comment field, I'm going to write up my first review comment.

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.

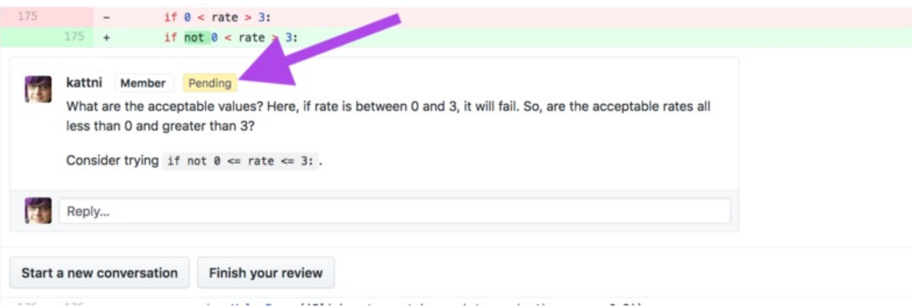
I'm going to ask about the acceptable values. In this case, I'm certain that it's not right as it is, and I have an idea of how to fix it. So, I'm also going to suggest a change to the code. GitHub supports Markdown in comments, so I'm going to format the code so the comment is easier to read.

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.

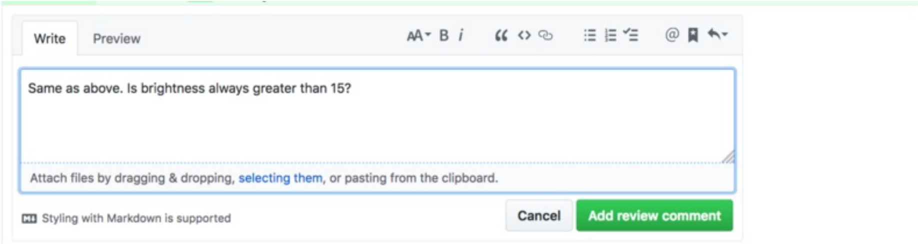


This begins my review by saving this first comment. It won't officially submit it until I'm ready. It will instead show my comment as **Pending** until I choose to complete the review.

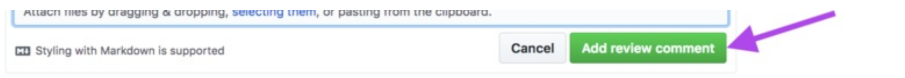


Now I'm going to repeat the process for the second issue. First mouse over any part of the line to bring up the blue plus sign.

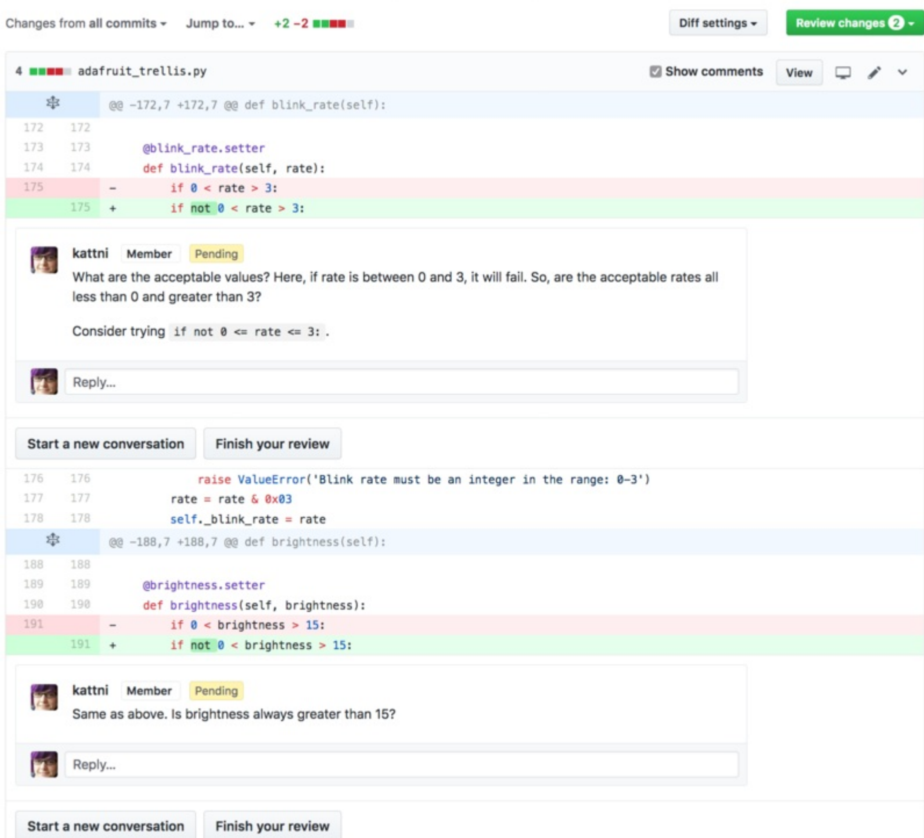
Click the blue plus to bring up the review comment window. Fill in your comment. In this case, since the issue is the same concept as the first, my comment is shorter because I can refer to the existing comment.



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

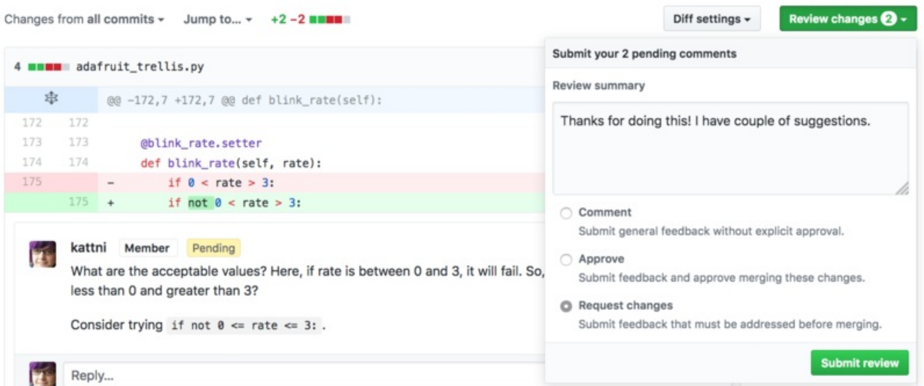


Now my second comment is pending as well. These are all the comments I would like to make to specific lines of code, so it's time to finalise my review.



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.

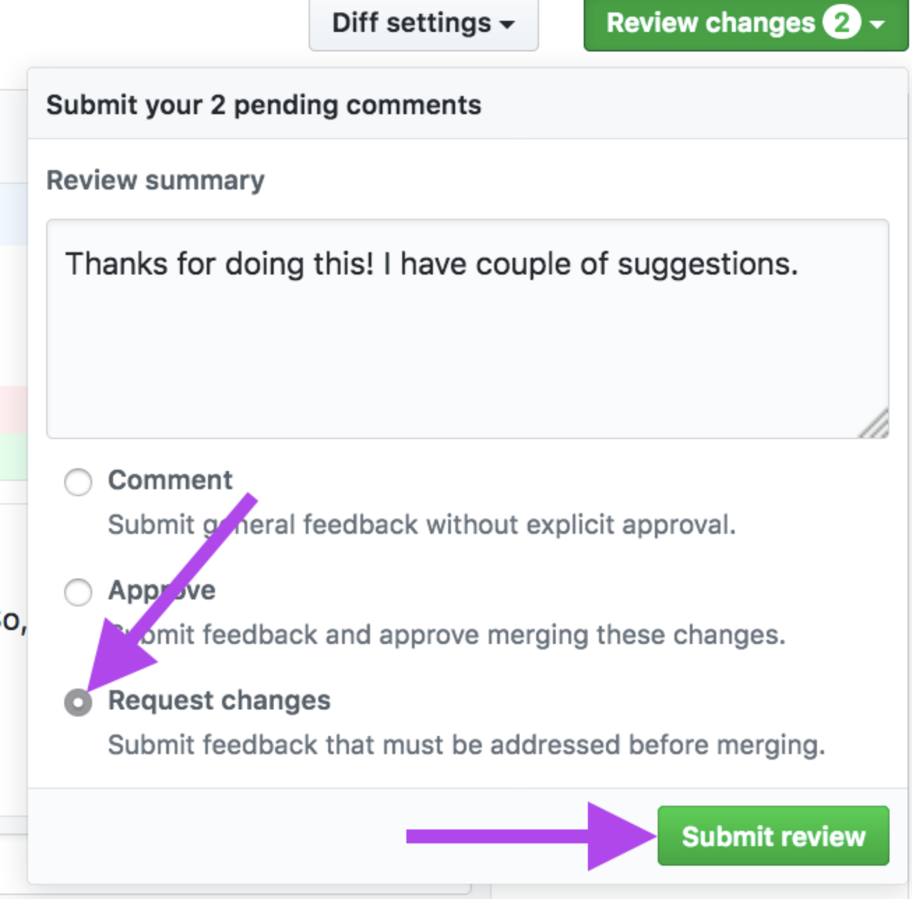
This will open window over the code with a comment field and some options below it. This is where you can add the main comment seen in the review. Whatever you put in this comment field will be at the top of your review, with your in-line comments below it. Here is where I'm going to thank Sommersoft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



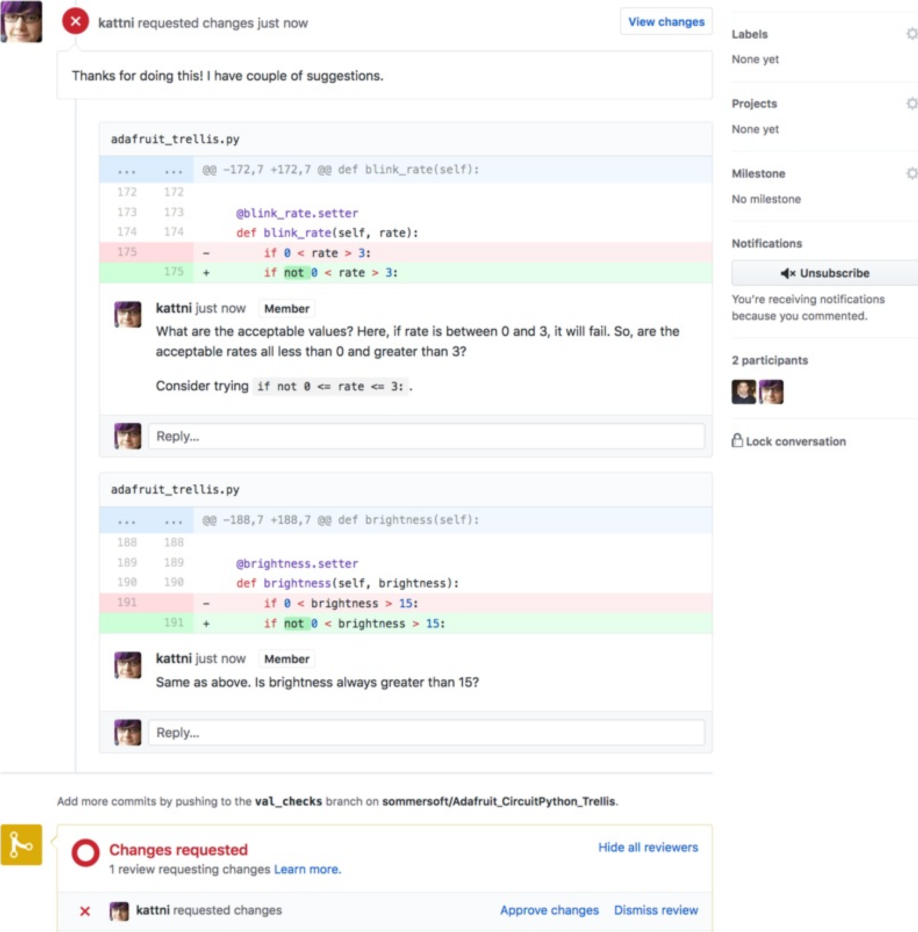
Notice that there are three options at below the comment box. It defaults to **Comment** which allows you tc submit a review containing only a comment if you choose. Below that is **Approve**, which you will choose when you're ready to approve the PR. Last is **Request changes**, which is what I've done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



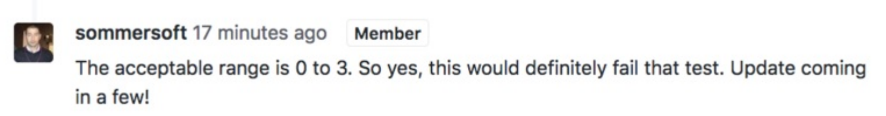
Now I'll be waiting for a response from Sommersoft.

Review Response and Update

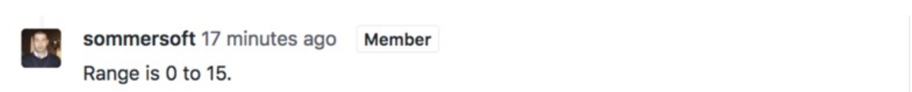
I received an email letting me know that there's been a response from Sommersoft on the PR. I'll navigate to the PR again, or refresh the page if I already have it open.

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.

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.



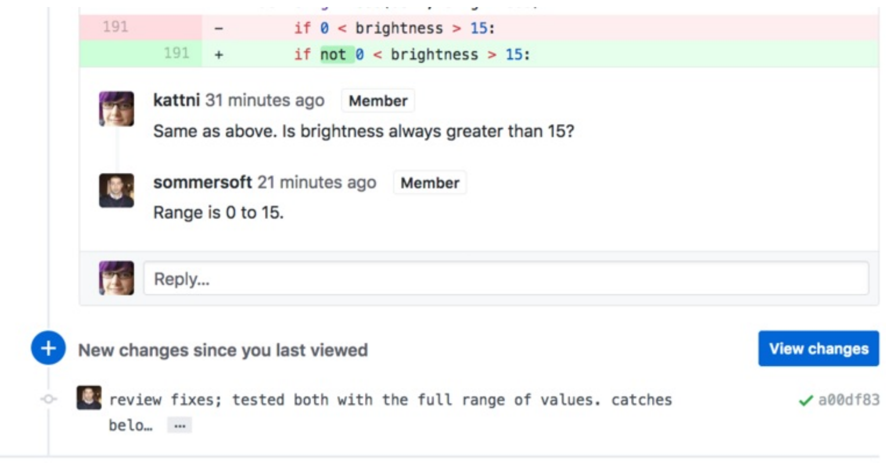
His second comment is similar to mine in that it simply answers the question asked. He's already let me know he will be submitting an update, so he hasn't reiterated it.



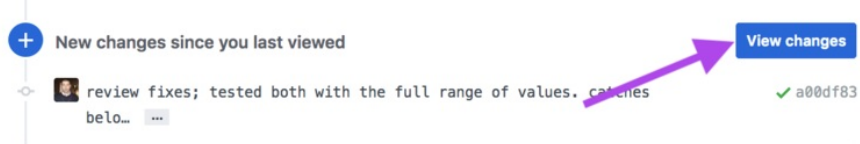
Great! I don't have anything further to add. Sommersoft understands the issues and will be fixing them, so now it's time for me to wait until he submits the fix.

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.



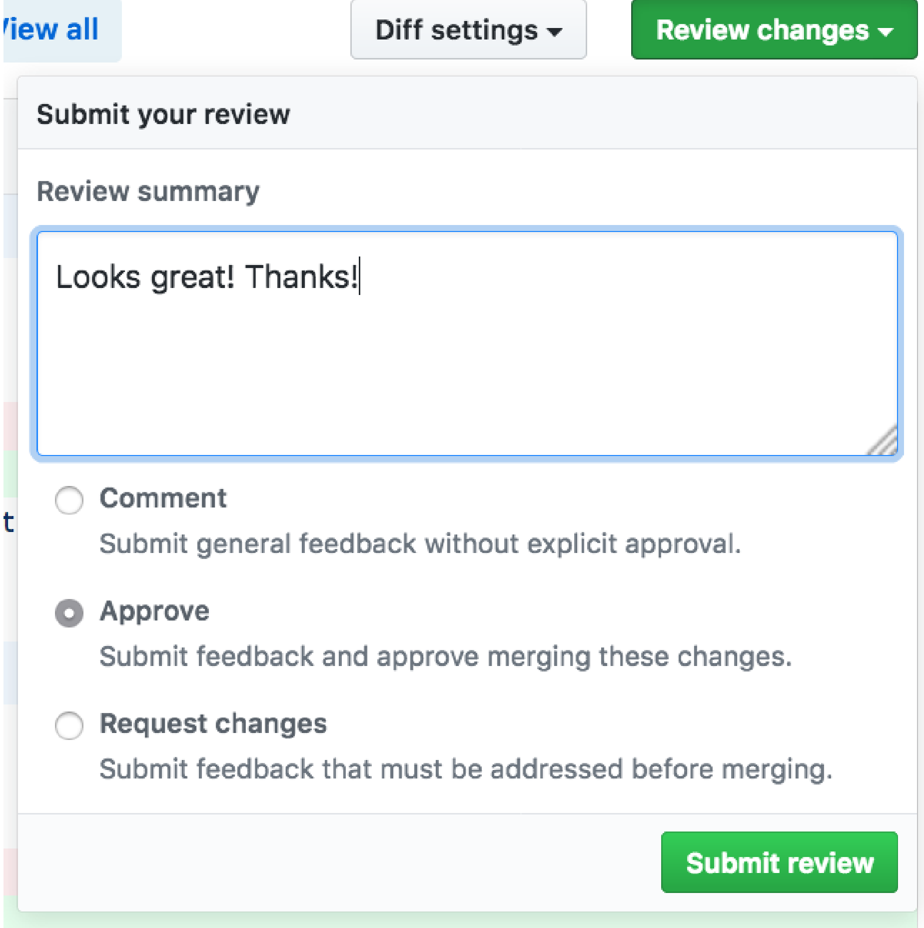
This will take you to the diff showing the most recent changes.

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.

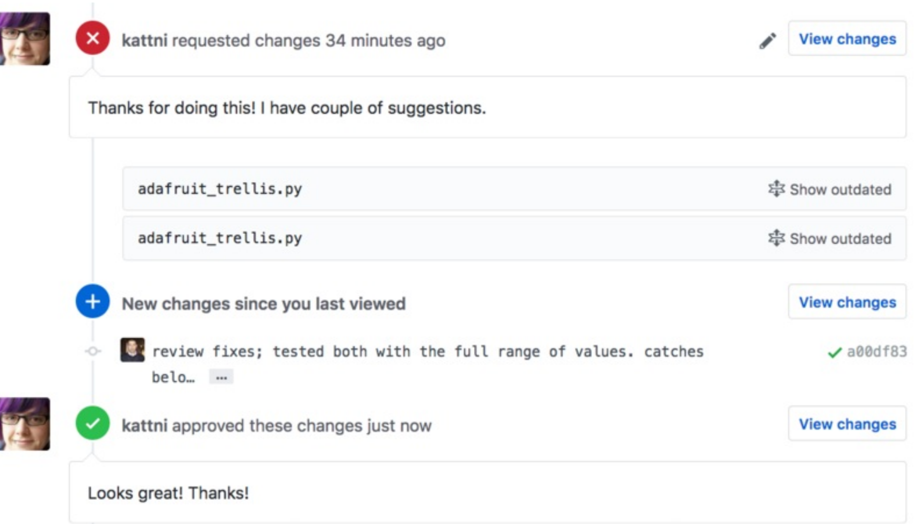
As you can see, Sommersoft took my suggestion for how to resolve the issue with the code. Excellent! I'm ready to approve the PR.

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**.



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.

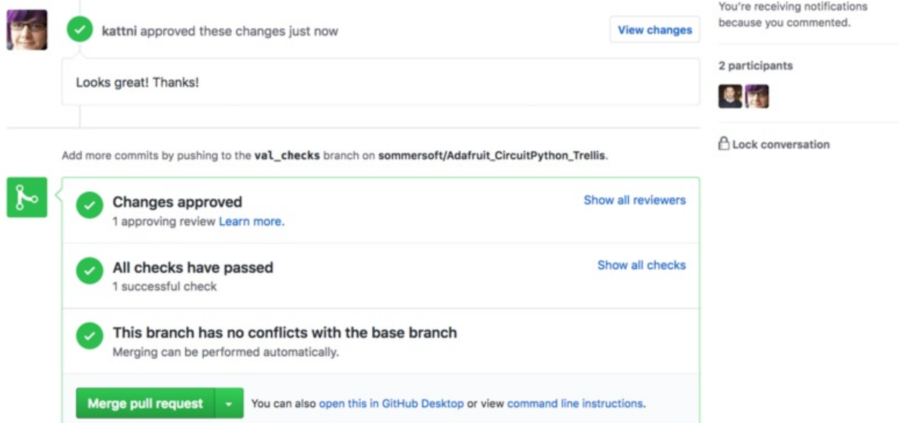


For those without write access to the repo, this is the end of the review process. This completes a significant portion of the entire review and greatly helps those of us with write access. We really

appreciate it!

Review Merge

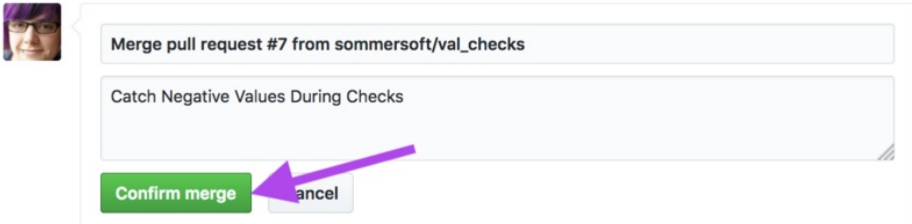
You've reviewed, requested changes, verified and approved the pull request. For those with write access to the repo, the final step is merging the PR. After you submit your review, the PR automatically returns to the conversation tab. You'll see your approval comment, and there should be green checks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



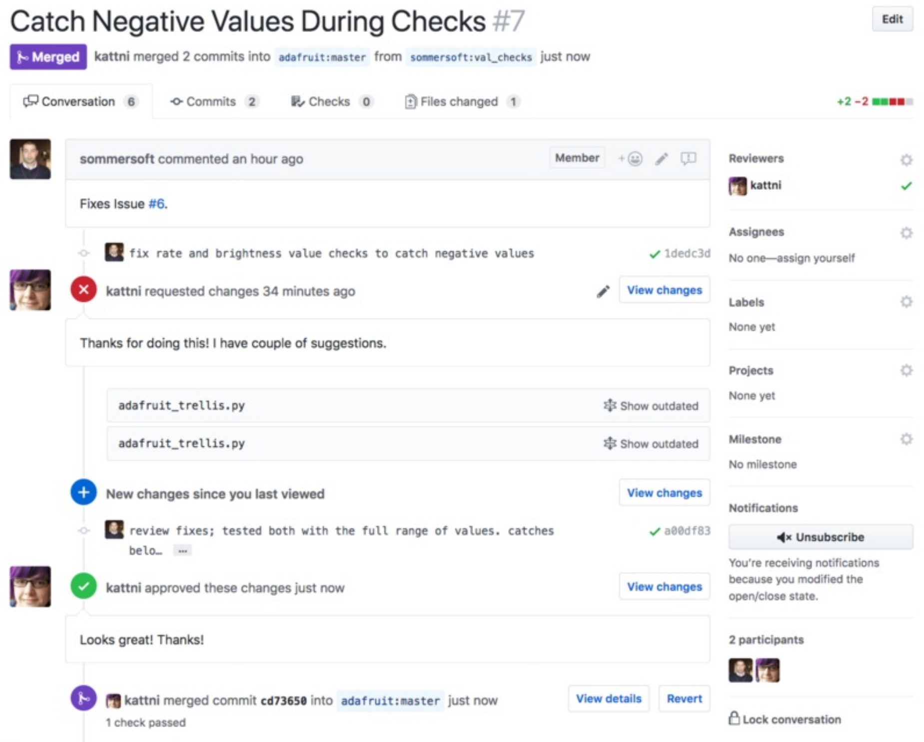
The **Merge pull request** button should also be green. Click the **Merge pull request** button.

This replaces the entire status section with the confirmation window. There's typically no reason to change any of the populated messaged 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 master 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!

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 main, default master branch clean. You can always merge your working branch changes into master at any time.

cd:

`cd` is the command to **c**hange **d**irectory 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.

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 master 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.

master:

The main, default branch is called master. It's good practice to make changes on a working branch and

leave the master 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 master 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 changed 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.