Contribute to the Adafruit Learning System with Git and GitHub

Created by Anne Barela


Last updated on 2022-03-04 03:14:59 PM EST
# Table of Contents

**Overview**
- Requirements 3
- The Adafruit Learning System 3
- Acknowledgements 4

**Workflow**
- If You Have a Repository (local copy) of the Adafruit Learning System already on your own computer 6

**Grab Your Fork**
- Clone Your Repo 7
- Remotes 9

**Always Work on a Branch**
- Starting from the Right Place 10
- Main or something else? 10
- Updating the Main Branch 11
- Create Your New Branch 12

**Status, Add, Commit, Push**
- Branched and Ready to Code 13
- Committing 13
- git status is Your Best Friend 13
- Time to commit 14
- Second commit and Further 16
- Push to Your Fork 17
- Not pushing to the Adafruit Learning System main Branch? 18

**Add Author and License Information**
- Arduino 18
- CircuitPython 19
- SPDX 19
- Troubleshooting 20

**Create Your Pull Request**
- Creating a Pull Request 20
- If you still don't see the Pull Request button 22
- Filling out the Pull Request 23
- If the CI Fails to Start 25

**Receiving a Review**
- Discussing the Review 27
- Submitting the Requested Changes 28
- Pull Request Approved 30

**Staying Up To Date**

**Glossary**

©Adafruit Industries
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. This guide focuses on the Adafruit Learning System specifically. There is also an excellent guide on contributing to CircuitPython (https://adafru.it/Vfq). This guide shares some common pages with that guide.

Requirements

Before starting this guide, there are a number of steps found in the Adafruit Learning System guide 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.

Much of this is covered in the Adafruit Guide An Introduction to Collaborating with Version Control (https://adafru.it/BHW).

Further information is available through the Git documentation (https://adafru.it/Bl1) or the GitHub documentation (https://adafru.it/Bl2).

This guide uses a terminal program to interact with Git locally (on your computer).

The Adafruit Learning System

This guide will walk through all of the steps I follow during the contribution process to the Adafruit Learning System. 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.

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/Bl0). If you're ever unsure about a term, feel free to look it up there.

Acknowledgements

This guide borrows very heavily on the guide Contribute to CircuitPython with Git and GitHub (https://adafru.it/Vfq) by Kattni Rembor who put a great deal of effort into explaining the process.

Also to Carter Nelson for the information flow diagram.

Finally, thank you Ladyada for your patience as I learned the processes.

Thank you all.

Workflow

The diagram below helps visualize where we will work on the process. Initially work will be in GitHub, getting a fork (copy) of the Adafruit Learning System repository
(repo for short). Then with git on your local computer, you will clone that fork, commit your additions, deletions, and edits, then push your changes back to your fork then finally create a Pull Request back to the Learning System repo.

I know the terminology may seem a bit strange - see the Glossary page in this guide for the meaning of each word.

This may help for the Adafruit Learning System:

The Repository (upstream) is the Adafruit Learning System on GitHub https://github.com/adafruit/Adafruit_Learning_System_Guides (https://adafrui.it/Clx)

The Repository (personal fork) is on your own GitHub account and is a copy of the Adafruit repository.

The Repository (local copy) is on your home or work computer. You will clone files from the Repository (personal fork) and push them back so you’re working within your own ecosystem and not Adafruit's.
If You Have a Repository (local copy) of the Adafruit Learning System already on your own computer

If you have previously done work on the Adafruit Learning System and you're going to do more work, please skip to and follow the page Staying Up To Date later in this guide to ensure you are working with the latest copy of the files. This will save many headaches if there is a conflict because your local copy was out of date compared to GitHub.

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 Learning System repository (https://adafruit.it/Clx).

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.

It should only take a few seconds.

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/TheKitty/Adafruit_Learning_System_Guides.git

(TheKitty is my GitHub username, yours will be different)

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

Now to fire up git on your local computer. This may vary by operating system, on Windows there is a Git Bash icon.
If you wish to create a new directory to place the files in locally, you might wish to do that prior to starting git.

Open your terminal program and navigate to your new directory using the `cd` command. I put my repos in a directory on my Windows S: drive named git:

```
Anne@ThreadRipper MINGW64 /s/git
$ cd /s/git
Anne@ThreadRipper MINGW64 /s/git
```

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 repos, Adafruit suggests setting the alias to the repo owner's GitHub user ID. In the case of this 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 (obtained above in GitHub):

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

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, /s/git/Adafruit_Learning_System_Guides, which contains a copy of the newly forked repo.

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

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.

Back in GitHub, 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 it is suggested 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 (for the Adafruit Learning System, it is adafruit), and entering the URL from GitHub obtained above:

```bash
git remote add adafruit https://original-project-url
```

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

Now you're ready to begin working with it!

---

**Always Work on a Branch**

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

**Starting from the Right Place**

Imagine you made a change to your code, but made a mistake. Now your repo is in a bad state. To help avoid this situation, we use branches. You always want to make changes while on a branch. A branch is a way to have your own working timeline of changes, while leaving the default branch even with the original project. The default branch is called main. It's best to leave main clean, and make your changes on a working branch.

For more details about branches, check out the [Branches? page](https://adafruit.it/BI4) found in the Adafruit guide [An Introduction to Collaborating with Version Control](https://adafruit.it/Blj).

**Main or something else?**

You may want to do is determine is whether the library you are working with is using main or something else as the default branch. This is a simple process. First, visit the library on GitHub. Above the repo contents on the left side is a drop down menu that shows all available branches.
It will typically be on the default branch to begin with, but you can verify by clicking on the menu.

Updating the Main Branch

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

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

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

```
git checkout main
```

There have been some updates to the remote main since I last did anything with this repo. Good thing I updated!
Now your main branch is even with the original project's main branch and you're ready to create your working branch!

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

Create Your New Branch

Now we can create a new branch. It's good practice to create a new branch for each new contribution you are working on. I'm working on adding license and author information to Learn guides, so I'll be doing all of it in one branch. However, if I intended to submit a fix for a specific guide 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 fixes to the Adafruit Learning System to add licensing information. So, I'm going to name my branch `license-changes`.

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
```

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

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

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

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

## 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.

### Commiting

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 utilize 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`.

As you can see, I've modified the file named Trinket_Gemma_Mini_theramin.ino (a project I did several years ago). 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.
I've added four 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`.

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 with a filename. To add all the files you changed, use `git add .`.

Followed by, you guessed it, `git status`.
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 Trinket_Gemma_Mini_Theramin.ino 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"
```

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`.

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
```
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 Adafruit Learning System **main** Branch?

As we create new releases and begin to work on new Learn projects, 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://adafruit.it/CXB).

---

**Add Author and License Information**

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

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

**Arduino**

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

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

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

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

### CircuitPython

CircuitPython only differs by using the Python comment `#` instead of the C comment `//` as depicted below:

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

An example:

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

### SPDX

Documenting information in this fashion is consistent with The Software Package Data Exchange® (SPDX®). SPDX is an open standard for communicating software bill of material information, including components, licenses, copyrights, and security references. SPDX reduces redundant work by providing a common format for companies and communities to share important data, thereby streamlining and improving compliance.
The SPDX specification is an international open standard (ISO/IEC 5962:2021 (https://adafru.it/VrB)).

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

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

Troubleshooting

- Be sure the comments are left justified and worded close to the example text including spaces.
- Be sure you used the right comment mark for the language you are coding with.
- Check that the license given is valid in the list here (https://adafru.it/VrC).

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.
If you still don't see the Pull Request button

Check that GitHub is registering that you have a commit ready (orange circle). This one does. So click on the Contribute Button (purple circle):

Then in the dropdown click on the Open Pull Request button:

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

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. First read the text that Adafruit has put for Adafruit Learning System PRs.

As I'm working on a defined project, I have added text describing what I have done.
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.

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. Below, I made one commit, changed one file, have no comments and I am 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.

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.

If the CI Fails to Start

If the GitHub Actions CI fails to process your PR, here is a simple way to trigger it. Go into your commit, edit one of the files ever so slightly - add a space (or delete one not needed) and it should trigger the CI to reexamine the PR again.

Note for Python files, do not add trailing spaces to a line, that will trigger an error.

Receiving a Review

The Continuous Integration (CI) checks on the 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`.

```
kattn@robotcave:Adafruit_CircuitPython_CircuitPlayground $ git diff
diff --git i/adafruit_circuitplayground/express.py w/adafruit_circuitplayground/express.py
index 3ef6f48..733f4f1 100755
--- i/adafruit_circuitplayground/express.py
+++ w/adafruit_circuitplayground/express.py
@@ -672,22 +672,18 @@ class Express:
     # pylint: disable=too-many-public-methods
     play_file("riasm0.wav")
     
-    # Play a specified file.
-    self.stng.on()
-    self.speaker_enable.value = True
-    try:
-        if sys.implementation.version[0] == 3:
-            file = audioio.WaveFile(open(file.name, "rb"))
-            audio.play(file)
-        elif sys.implementation.version[0] == 2:
-            audio = audioio.AudioOut(board.SPEAKER)
-            wavfile = audioio.WaveFile(open(file.name, "rb"))
-            audio.play(wavfile)
-            white.audio.playing;
-        audio.deinit()
-    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.
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 main 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**

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

The first thing you want to do is return to the main branch. If you're unsure which branch you're currently on, type `git branch` to see a list. The current branch will be highlighted with an asterisk next to it. You should still be in the branch you created. Now, let's return to main.
To check out the main branch, enter the following `checkout` command:

```
$ git checkout main
```

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 newest version of the remote repo, but does not merge it into the current repo.

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

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

```
$ git fetch ownerid
```

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

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

```
$ git merge ownerid/main
```

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

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

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

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

```
git push yourid main
```

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

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

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

add:

`add` is the command used to stage a changed file for commit. When you add a file, it changes the status from **Changes not staged for commit:** to **Changes to be committed:**. This means when you next commit, any files you `add` will be included.

branch:

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

cd:

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

change request:

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

checkout:

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

clone:

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

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

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

continuous integration testing:

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

diff:

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

When you view the diff as part of a pull request, it shows you all the changes included in that PR. It also only shows you the code around the changes to conserve space.
fetch:
Fetching is the act of grabbing the changes from a remote repo, but not merging them in. You'll use `fetch` when you're preparing to update your main branch to be in line with the original project.

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

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

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

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

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

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

push:

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

remote:

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

repository or repo:

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

review:

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

---

**staged:**

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

---

**status:**

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

---

**Travis CI:**

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

---

**upstream:**

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