Merging Branches with Git

[(https://github.com/learn-co-curriculum/git-github-merging-branches)](https://github.com/learn-co-curriculum/git-github-merging-branches) [(https://github.com/learn-co-curriculum/git-github-merging-branches/issues/new)](https://github.com/learn-co-curriculum/git-github-merging-branches/issues/new)

Learning Goals

- Perform standard and fast forward merges with the `git merge` command.
- Explain the difference between standard merges and fast forward merges.
- Explain two other techniques that can be used when incorporating changes: `squash` and `rebase` .

Introduction

In the last lesson, we learned how to create and use branches so we can work on new features without affecting the code in the `main` branch. The next step is to learn how to incorporate those changes into the central code base once we're ready. In this lesson, we'll learn to perform merges using the `git merge` command. We'll also learn how different types of merges affect our Git history. Finally, a couple of additional techniques that give us greater control over the commit history will be introduced.

Merging Branches

To merge one branch into another, the steps are:

1. Switch to the receiving branch, i.e., the one you want to merge the other branch into. This is frequently `main` , but it can be any branch in your project.
2. Run `git merge <branch-to-merge>` .

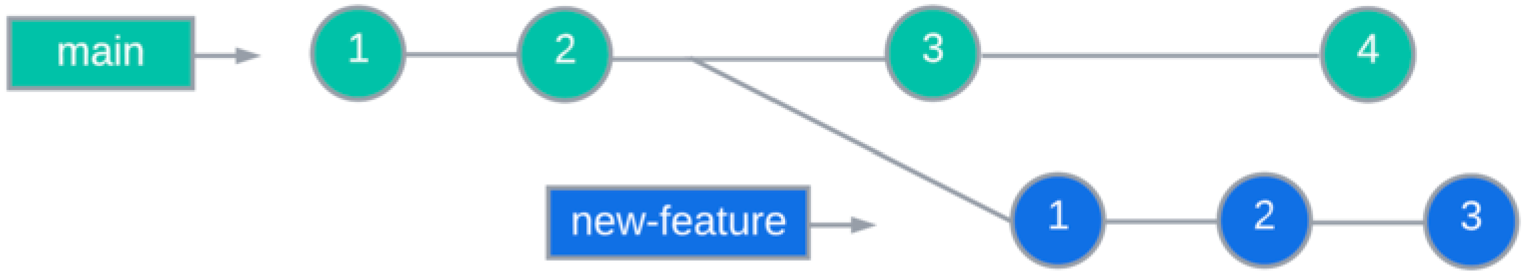
So, for example, if we had a `bug-fix` branch we were ready to merge into the `main` branch, we would switch to `main` then run:

```
$ git merge bug-fix
```

That's it. If there aren't any conflicts (more on this in the next lesson), Git will merge all the changes you made in `bug-fix` into the code and update `main` 's Git history to include the commits that were made on the `bug-fix` branch. What that history winds up looking like depends on what's in the commit history of the two branches.

Standard Merge

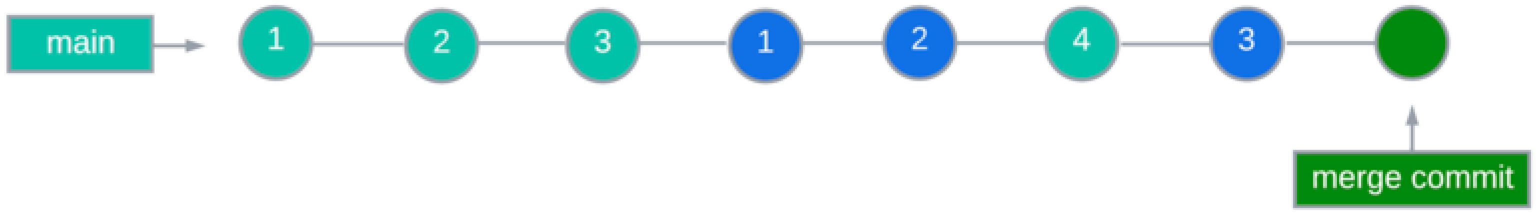
Let's say the status of our repo is the following:



We have created a `new-feature` branch and have made three commits on that branch. In the meantime, two additional commits have been made on the main branch. When we run `git merge new-feature` , Git will do two things:

1. It will combine the commits from both branches into the main branch's Git history, in order of when they were made.
2. It will create an additional commit, a *merge commit*, and add that to the end of the commit history.

It would look like this:

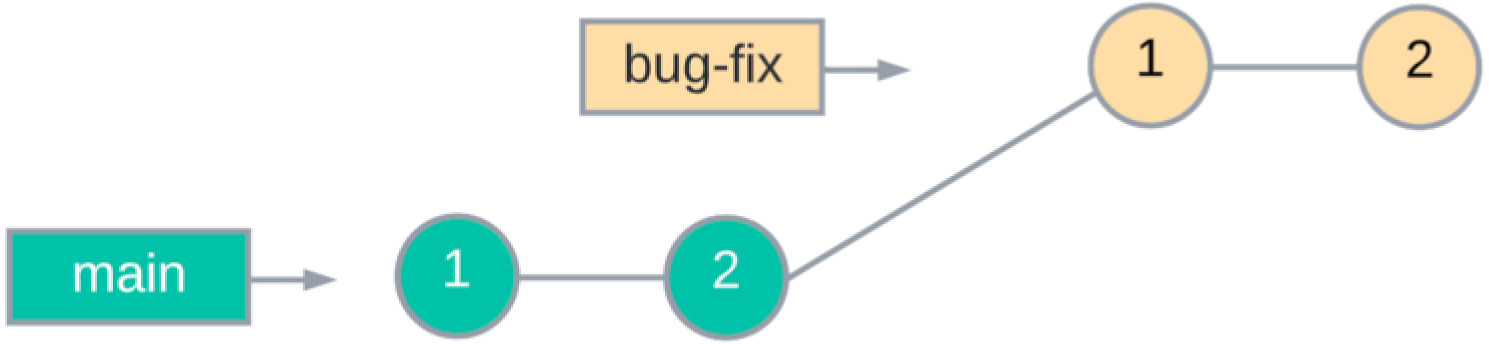


When you run the command, Git opens a file in the Git command line editor so you can enter a commit message for the merge commit. You can add a message, save the file and close it to complete the merge, or you can just close the file if the default message is adequate.

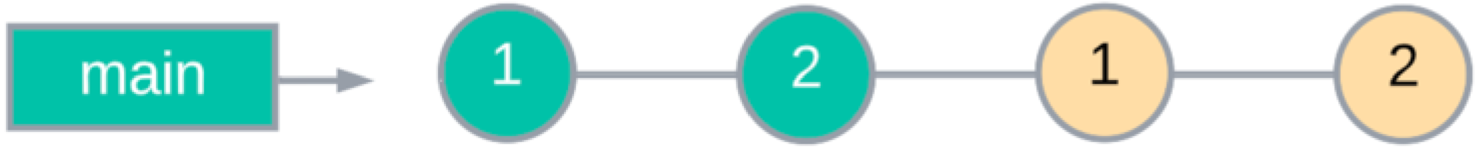
The advantage of this type of merge is that it retains the full history of commits from both branches, and retains the order in which the commits were made. The disadvantage is that it adds a new, "empty" commit, that just indicates when the two branches were merged. It will also show the commits for `main` and for the feature you added interspersed in the history, which can make it harder to follow. You can imagine that, if you're working on a very active project with lots of branches being created and merged in, the Git history could get quite long and complicated.

Fast Forward Merge

A fast forward merge is just a simplified version of the standard merge, in which the history of the receiving branch has not changed since the second branch was created:



In this case, the merge simply appends the commits from `bug-fix` to the end of the main branch's history. No merge commit is necessary, because it is clear from the commit history when `bug-fix` was merged in:



Because nothing has changed on the main branch since we created the branch, when we merge we are simply "fast forwarding" the main branch to the current state of the feature branch.

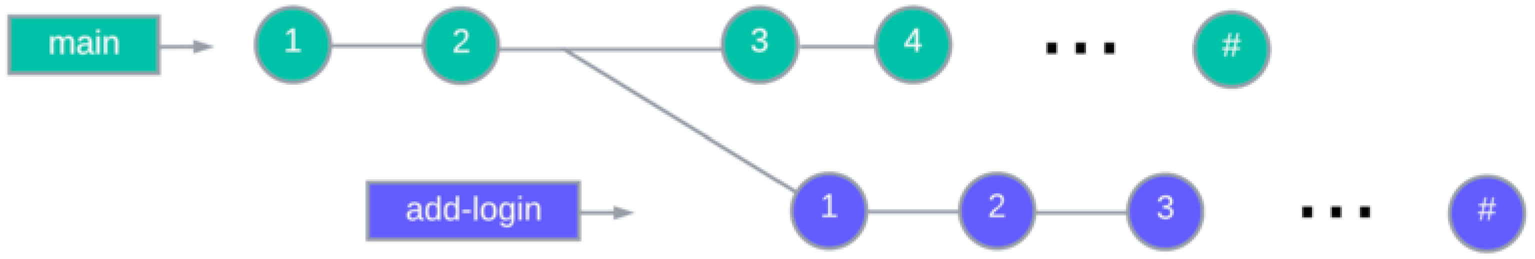
Squashing

There are a couple of other options Git provides to help keep your Git history cleaner and easier to read; the first of these is `squash`.

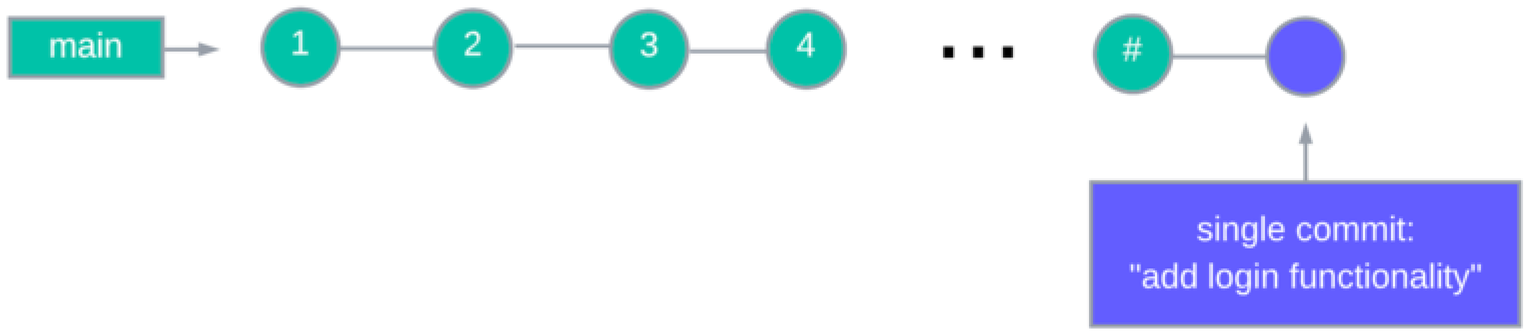
Say we've created an `add-login` branch where we've developed login functionality for our application. Our branch contains quite a few commits, one for each discrete step we completed to get it working. We've thoroughly tested our code, and we're ready to merge it in. However, we know if we use a standard merge, all those commits will be interspersed among the commits on our `main` branch, and a merge commit will be added. This will make our Git history harder to read and full of a lot more detail than we really need.

What `squash` allows us to do is "squash" all the commits on the `add-login` branch together into a single commit on our `main` branch. With this approach, the addition of the feature is documented without cluttering up our history.

Before the merge:



After the merge:



The main downside of using squash is that the history of the commits on the `add-login` branch will be lost. Whether this is an acceptable tradeoff depends on your needs.

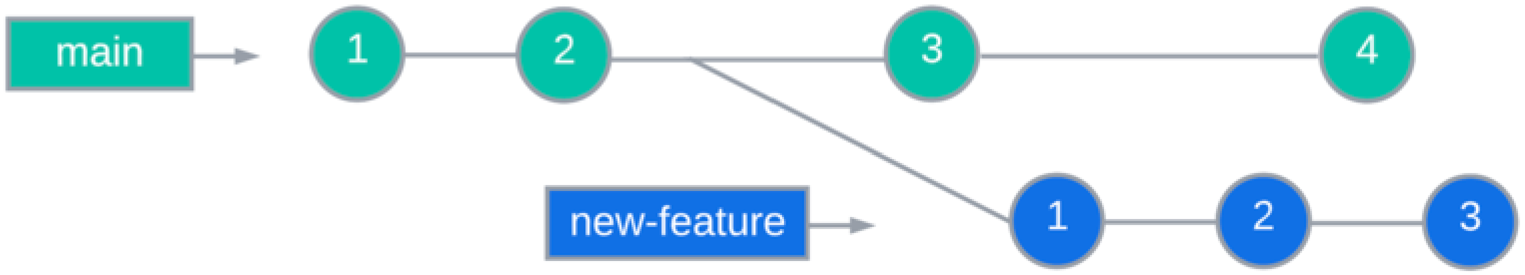
We will see an example using `squash` in a later lesson.

Rebasing

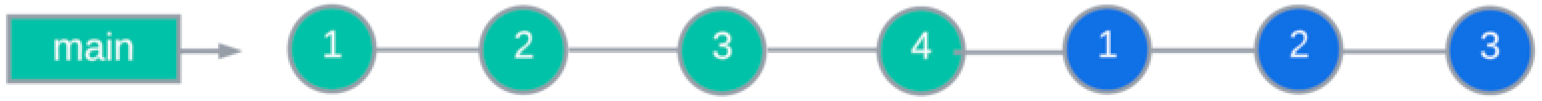
Another option you can use to keep your Git history cleaner is `git rebase`. This command can be used either after the fact (i.e., after merging) to clean up your commit history, or as an **alternative to merging (https://www.atlassian.com/git/tutorials/merging-vs-rebasing)**.

When `rebase` is used in place of `merge`, it "rewrites" history by moving (rebasing) the point at which the feature branch was created to the end of the main branch's history. As a result, the history shows all the commits on the main branch, followed by all the commits from the feature branch, making it look similar to a fast forward merge.

If we use the `new-feature` example from above:



The result after rebasing would look like this:



The advantages of rebasing are that it makes it easier to see the history of different features, since those commits are kept together, and it removes the need to create *merge commits*. The disadvantage is that the order of the commits no longer reflects when they were made — you have "rewritten" history.

Warning: Rebasing is generally considered to be an "advanced" technique, because rewriting your Git history can cause problems, especially when you're working as part of a team. If you rebase your Git history and your collaborators have an earlier version of the code with a different history, it can be quite a mess to clean up! For this reason, we recommend getting comfortable with merging and with the collaborative workflow before delving into rebasing.

Exercise: Merging the changes in our feature branch

In the previous lesson, we added some books to our Python study resources, then created a branch to make the book titles into links. Our next step will be to `merge` our changes into the `main` branch.

Note: before continuing, make sure the state of your project matches where we left off at the end of the last lesson (after the exercise).

On the `main` branch, the file should look like this:

```
# Python Study Resources

## Books

- Composing Programs (Chapter 1-2) [Free]
  - This is a great book for learning compsci concepts in addition to Python
- Automate the Boring Stuff with Python [Free]
  - Has short, easy projects that involve different modules
- Python Crash Course , 2nd Ed
  - Beginner friendly intro to fundamental concepts
```

And the commit history should look like this:

```
commit 79971225f5084ac808dc6fc39a385acf45071fc5 (HEAD -> main)
Author: Your Name <youremail@email.com>
Date:   Sat May 7 19:32:26 2022 -0400

    add books to list

commit 1b3142a6c7ebec9001f2fe4a2436746d9b076a49
Author: Your Name <youremail@email.com>
Date:   Sat May 7 19:29:21 2022 -0400

    initial commit
```

If you switch to `add-links` , they should look like this:

```
# Python Study Resources

## Books

- [Composing Programs (Chapter 1-2)](https://composingprograms.com/) [Free]
```

```
    - This is a great book for learning compsci concepts in addition to Python
  - [Automate the Boring Stuff with Python](https://automatetheboringstuff.com/) [Free]
    - Has short, easy projects that involve different modules
  - [Python Crash Course , 2nd Ed](https://nostarch.com/pythoncrashcourse2e)
    - Beginner friendly intro to fundamental concepts

commit 75a9bdbcea69c1de18453ad6b89b4819b2ad10b4 (HEAD -> add-links)
Author: Your Name <youremail@email.com>
Date:   Sat May 7 19:38:47 2022 -0400

    add links for book titles

commit 79971225f5084ac808dc6fc39a385acf45071fc5 (test, main)
Author: Your Name <youremail@email.com>
Date:   Sat May 7 19:32:26 2022 -0400

    add books to list

commit 1b3142a6c7ebec9001f2fe4a2436746d9b076a49
Author: Your Name <youremail@email.com>
Date:   Sat May 7 19:29:21 2022 -0400

    initial commit
```

If anything in your code doesn't match what's shown above, we recommend revisiting the lesson on undoing changes to figure out how to fix it. It will be good practice!

Once you're ready, go ahead and merge the `add-links` branch into main:

1. Make sure you're on the `main` branch.
2. Run the `git merge` command.
3. Verify that the file on the `main` branch has been updated with the links.

When you're done, the commit history on the `main` branch should be the same as the commit history on the `add-links` branch.

Make sure you have successfully merged the `add-links` branch before moving on. We will continue working with this project in the next lesson.

Check for Understanding

Before moving on to the next lesson, check for your understanding of this material by answering the following questions in your own words:

- What is the difference between a standard merge and a fast forward merge?
- What do we mean when we say that rebasing rewrites the history of a repo?

Conclusion

Which technique you choose for incorporating the changes in one branch into another will depend on your needs. If it's more important to maintain a complete history of all the changes and commits, a standard merge is probably the way to go. If, on the other hand, if you want your commit history to provide concise, readable documentation of how your code has changed and expanded over time, you may want to use one of the more "advanced" techniques, such as rebasing.

Rebasing is a very powerful tool that is used frequently by many people. However, it can also cause problems, especially when you're working as part of a team. We have introduced it here so you are aware of it and what it can do for you, but we recommend that you focus on getting more comfortable with using branches for now.

Resources

- **Different Merge Types in Git (https://lukemerrett.com/different-merge-types-in-git/)**
- **Merging vs. Rebasing (https://www.atlassian.com/git/tutorials/merging-vs-rebasing)**