Viewing the Git Commit History

(https://github.com/learn-co-curriculum/git-github-viewing-git-history) (https://github.com/learn-co-curriculum/git-github-viewing-git-history/issues/new)

Learning Goals

- Use `git diff` to view changes in a project's history.
- Use `git log` to view commits.

Introduction

In earlier lessons, we discussed what a commit is, how to commit our work, and how to write good commit messages. Following good practices for making commits means that we have a complete and accurate history of the status of our code. But Git also allows us to interact with our commits in various ways to get detailed information about what changes were made and when, as well as to undo or amend those changes. In this lesson, we will learn how to use `git diff` (https://git-scm.com/docs/git-diff) and `git log` (https://git-scm.com/docs/git-log) to explore our Git history. Then, in the next lesson, we'll learn some Git commands that can be used to undo changes.

Diffing

You can use Git's `diff` command to view the changes you have made to your code. Git enables us to compare almost anything you can imagine: you can look at the differences between different commits or different branches, and between code at different stages in our working directory.

Comparing File States in the Working Directory

Recall that, when you run `git status`, files in the working directory can be in one of three states, represented in the diagram below:

1. They can have *untracked changes* that have not yet been *added*.
2. They can have *staged changes* that have been added but not yet committed.

3. They can be *clean*, with no changes since the last commit.

 Basic git workflow

The `git diff` (https://git-scm.com/docs/git-diff) command gives us the ability to compare code across these three different states in various ways. To do this, there are three variations of the `git diff` command that are commonly used:

- `git diff` compares the working directory (unstaged changes) to the next most recent version of the file: either the staged changes (if there are any), or the last commit.
- `git diff --cached` or `git diff --staged` - these both do the same thing: compare staged changes to the latest commit (middle vs. bottom).
- `git diff HEAD` - compares *all* uncommitted changes (both staged and unstaged) to the latest commit (top plus middle boxes vs. the bottom box).

> **What is the HEAD?** The term `HEAD` is a nickname that usually references a particular branch (in our case, `main`), and, more specifically, the most recent commit on that branch, which is the commit we are most often interested in looking at. However, it is possible to *check out* a specific commit, which updates the working directory to show the code as it was at that particular point in our commit history. In this case, the `HEAD` nickname would be pointing to **that** commit.
>
> This situation is referred to, somewhat alarmingly, as a "detached HEAD" state, but it just means that `HEAD` is no longer pointed to the latest commit on our active branch. For now, you can just think of `HEAD` as a reference to the latest commit in the current working branch.

Diffing Example: Backyard Bird List

Let's say that we've recently become interested in birdwatching (or, to use the preferred term, *birding*), and we want to start keeping a list of all the birds that we see in our yard. Go ahead and make a directory for our new project, cd into it, and initialize it as a repository:

```
$ mkdir backyard_bird_list
$ cd backyard_bird_list
$ git init
Initialized empty Git repository in <path-to-project>/backyard_bird_list/.git/
```

So far all we've done is create an empty directory, so if we run `git status` we should see that there's nothing to commit yet:

```
$ git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Let's go ahead and create a file to store our list in:

```
$ touch bird_list.md
$ git status
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    bird_list.md

nothing added to commit but untracked files present (use "git add" to track)
```

We're creating a **markdown (https://docs.github.com/en/get-started/writing-on-github/getting-started-with-writing-and-formatting-on-github/basic-writing-and-formatting-syntax)** file (indicated by the `.md` extension) so we can use GitHub markdown to add some formatting to our file. Markdown provides a simple syntax for adding common formatting (bold or italics, headings, links, images, ordered and unordered lists, etc.) to a text file without having to write HTML.

Next we'll stage our new file and make our initial commit:

```
$ git add bird_list.md
$ git commit -m 'initial commit'
[main (root-commit) 416964f] initial commit
```

```
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 bird_list.md
```

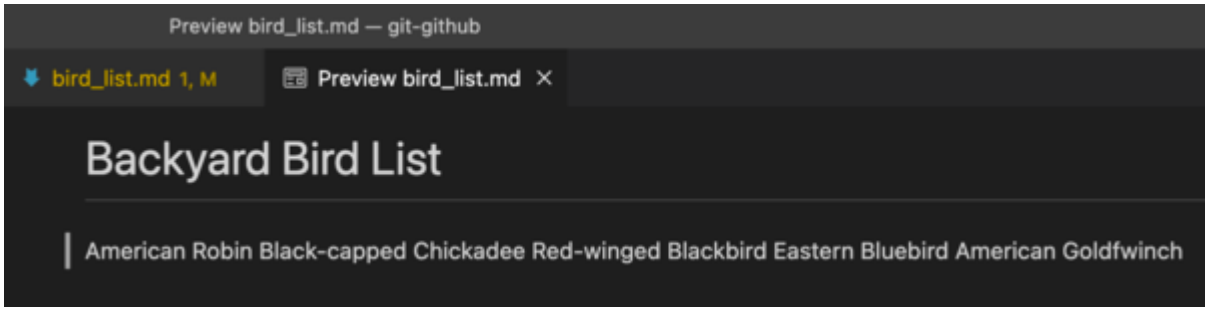
Now we're ready to start adding our birds. Open the project in VS code, then add the following to `bird_list.md` and save the file:

```
# Backyard Bird List

American Robin
Black-capped Chickadee
Red-winged Blackbird
Eastern Bluebird
American Goldfwinch
```

In markdown, we use the `#` to indicate that the text that follows should be the top-level header for our document. Make sure there's a space between the `#` and the text. For a second-level header, we would use `##`, and so on, up to six levels.

Most text editors enable you to look at a preview of what your markdown file will look like. In VS Code, you can do this by right-clicking on the name of the file — either in the tab at the top of the text editor window or in the file list on the left — and selecting Open Preview. If you do this now, you'll see that the bird names appear in one long line, rather than on separate lines:

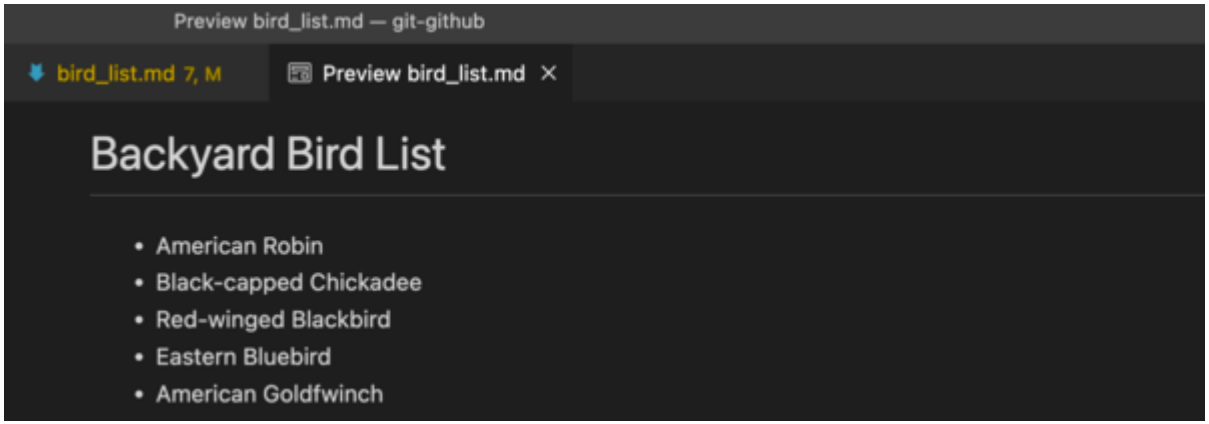


Definitely not what we wanted. We need to indicate somehow that we want our list formatted as a list. We do that by prepending each bird name with either `*` or `-`, followed by a space. Which one you choose is a matter of preference, but you should try to be consistent. Let's update our file to do that:

```
# Backyard Bird List

- American Robin
- Black-capped Chickadee
- Red-winged Blackbird
- Eastern Bluebird
- American Goldfwinch
```

If you check the preview again, it should now look like this:



Much better!

Reading the Ouput of `git diff`

Next, let's run `git diff` ; you should see something like the following:

```
$ git diff
diff --git a/bird_list.md b/bird_list.md
index e69de29..6b163d7 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -0,0 +1,7 @@
+# Backyard Bird List
```

```
+
+- American Robin
+- Black-capped Chickadee
+- Red-winged Blackbird
+- Eastern Bluebird
+- American Goldfwinch
```

Hmm, there's a lot to look at here. Let's take it line by line:

```
diff --git a/bird_list.md b/bird_list.md
```

This line indicates what two file versions are being compared: here, we're comparing `bird_list.md` at two different points in time. The identifier `a` refers to the earlier version of the file (from our initial commit), and `b` refers to the later version of the file (the unstaged changes currently in our working directory).

The next line (`index e69de29..6b163d7 100644`) contains some metadata that you can disregard.

```
--- a/bird_list.md
+++ b/bird_list.md
```

These two lines provide a legend for the diff output: lines marked with `-` contain something that was in the earlier version of the file (`a`) but has been removed or changed, and lines marked with `+` contain something that has been added or updated and is only in the newer version of the file (`b`).

```
@@ -0,0 +1,7 @@
```

This line indicates what lines in the `a` (`-0,0`) and `b` (`+1,7`) files are represented in the diff output. Inside each set of parentheses, the first number in the pair is the line number of the first line shown, and the second is the total number of lines shown. In this case, because the `a` version of the file was empty, it lists `-0,0` . For the later version of the file, however, `+1,7` indicates that the diff output shows a total of 7 lines, starting with line 1. For this example, these line numbers are not particularly useful, but they will be when you have files that contain dozens or hundreds of lines of code.

Finally, we see the diff itself:

```
+# Backyard Bird List
+
+- American Robin
+- Black-capped Chickadee
+- Red-winged Blackbird
+- Eastern Bluebird
+- American Goldfwinch
```

This is showing us that no lines in the earlier version of the file were deleted or changed (not surprising, given that the file was empty), and that 7 lines were added and are present in the current version of the file.

Recall that `git diff` shows us the difference between the unstaged changes and the next most recent version of our file — in this case, our initial commit. If we go ahead and stage our changes:

```
git add bird_list.md
```

then run `git diff` again, nothing will be output because we don't have any unstaged changes. If we want to see the difference between our *staged* changes and the most recent commit, we can use `git diff --staged` ; in this case, the output will look the same as before.

Our list is now staged and ready to commit, but we just realized that we have a typo. Go back into VS Code and remove the "w" from "Goldfwinch", so it reads "Goldfinch" instead, then save the file. Once you've done that, run `git diff` again:

```
$ git diff
diff --git a/bird_list.md b/bird_list.md
index 5807b4a..b03ebf1 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -4,4 +4,4 @@
 - Black-capped Chickadee
 - Red-winged Blackbird
 - Eastern Bluebird
```

```
-- American Goldfwinch
+- American Goldfinch
```

This output is showing us the difference between our unstaged changes and what is now the next most recent version of the file: the staged changes. The last two lines indicate that `"- American Goldfwinch"` has been removed from the file and `"- American Goldfinch"` added in its place.

If we want to see the difference between **all** our uncommitted changes (staged and unstaged) and the most recent commit, we can run `git diff HEAD`:

```
$ git diff HEAD
diff --git a/bird_list.md b/bird_list.md
index e69de29..b03ebf1 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -0,0 +1,7 @@
+# Backyard Bird List
+
+- American Robin
+- Black-capped Chickadee
+- Red-winged Blackbird
+- Eastern Bluebird
+- American Goldfinch
```

Now we can see that, once we've committed all of our changes, the `bird_list.md` file will include all the birds that we added, without the typo.

At this point, there are two ways we could proceed: we could go ahead and commit our staged changes, then add and commit the typo fix, **or** we can re-add the file first and do a single commit. There's no real reason to make a separate commit for the typo fix — that isn't information we're likely to care about in the future — so let's keep our commit history a bit cleaner and re-add `bird_list.md` then commit our changes:

```
$ git add bird_list.md
$ git commit -m "add first day's birds to list"
```

```
[main 7cff5a0] add first day's birds to list
 1 file changed, 7 insertions(+)
```

Now if you run `git status`, you should see that the working directory is clean and there's nothing to commit.

> Question: what will we see if we run one of the `git diff` commands now?

Comparing Specific Commits

We've seen how `git diff` allows us to compare the different states of the code in our working directory, but it can also be used to compare **any** two commits by simply specifying the two SHA's:

```
$ git diff commit1-sha commit2-sha
```

We can use either the full SHA or the abbreviated version.

If one of the two commits we are comparing is HEAD, we can use the nickname in place of the SHA:

```
$ git diff commit1-sha HEAD
```

You can provide the two commits in whatever order makes the most sense to you but, by convention, file 'a' (the first one listed) usually represents an earlier point in time than file 'b'. Recall, for example, that when we ran `git diff` earlier, file 'a' referred to HEAD (the initial commit), while file 'b' referred to the changes we made after that commit. To avoid confusion, it's a good idea to follow this convention when you specify the commits that are to be compared.

Let's compare the two commits we've made so far for our backyard bird list. If we scroll back up through the output in our terminal (don't worry — we'll learn about a better way shortly), we can find the abbreviated SHA for the initial commit as well as the second commit. In our case (remember: yours will be different), the SHA for the initial commit was `416964f` and the SHA for the second commit was `7cff5a0`, so to compare the two, we would run the following:

```
$ git diff 416964f 7cff5a0
```

Or we can save ourselves a little bit of hunting by using the `HEAD` nickname in place of the SHA For the second commit:

```
$ git diff 416964f HEAD
```

With either option, the result will show that we started with an empty file in our initial commit, and added 7 lines to it in our second commit:

```
diff --git a/bird_list.md b/bird_list.md
index e69de29..dd3abf2 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -0,0 +1,7 @@
+Backyard Bird List
+
+American Robin
+Black-capped Chickadee
+Red-winged Blackbird
+Eastern Bluebird
+American Goldfinch
```

Comparing Branches

Finally, you can use `git diff` to compare any two *branches*:

```
$ git diff branch1 branch2
```

We'll talk more about this use of `git diff` when we get to the lessons on branching.

Git Log

Running `git log` inside your working directory will print a list of all the commits that have been made. If you run it inside your `backyard_bird_list` directory, you should see something like this:

```
$ git log
commit 7cff5a0e0e002a61317b7d3c5282c042e8de0b06 (HEAD -> main)
Author: Your Name <youremail@email.com>
Date:   Fri Apr 22 17:35:10 2022 -0400

    add first day's birds to list

commit 416964f85cd5cb813500ca713aab806e54f94a08
Author: Your Name <youremail@email.com>
Date:   Fri Apr 22 16:06:43 2022 -0400

    initial commit
```

The most recent commit is listed first, followed by each earlier commit in reverse chronological order. You'll see that the first commit is where HEAD is currently pointing: commit 7cff5a0.... on the `main` branch.

Let's go ahead and add our next day's birds to the bottom of `bird_list.md` :

```
...
- White-breasted Nuthatch
- Tufted Titmouse
- Carolina Wren
```

For practice, let's run `git diff` :

```
$ git diff
diff --git a/bird_list.md b/bird_list.md
index b03ebf1..bfe4c22 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -5,3 +5,6 @@
 - Red-winged Blackbird
```

```
 - Eastern Bluebird
 - American Goldfinch
+- White-breasted Nuthatch
+- Tufted Titmouse
+- Carolina Wren
```

The line number information (`@@ -5,3 +5,6 @@`) is telling us that the output shows 3 lines from file `a` and 6 lines from file `b` , both starting at line 5. No lines of code were removed or changed, and 3 lines were added. This looks like what we wanted to do, so let's add and commit our changes:

```
$ git add bird_list.md
$ git commit -m "add second day's birds to list"
```

Now, if we run `git log` again, we'll see all three commits, with the last one listed first:

```
$ git log
commit f9658e3db05e5d7894053c9cc44d91182d2918ed (HEAD -> main)
Author: Your Name <youremail@email.com>
Date:   Fri Apr 22 21:33:58 2022 -0400

    add second day's birds to list

commit 7cff5a0e0e002a61317b7d3c5282c042e8de0b06
Author: Your Name <youremail@email.com>
Date:   Fri Apr 22 17:35:10 2022 -0400

    add first day's birds to list

commit 416964f85cd5cb813500ca713aab806e54f94a08
Author: Your Name <youremail@email.com>
Date:   Fri Apr 22 16:06:43 2022 -0400
```

```
    initial commit
```

Here is where developing good habits around writing commit messages comes in handy: if we wanted to see what birds we added to our list on the first day, for example, we know we need to look at the difference between the initial commit and the commit with the message "add first day's birds to list":

```
$ git diff 416964f 7cff5a0
diff --git a/bird_list.md b/bird_list.md
index e69de29..dd3abf2 100644
--- a/bird_list.md
+++ b/bird_list.md
@@ -0,0 +1,7 @@
+Backyard Bird List
+
+American Robin
+Black-capped Chickadee
+Red-winged Blackbird
+Eastern Bluebird
+American Goldfinch
```

And here we see the 5 birds we added the first day.

A Couple of Helpful `git log` Options

Like most git commands, **git log (https://git-scm.com/docs/git-log)** has a large number of options you can use with it. In our case, because we only have a few commits, the output of our `git log` isn't difficult to use. But if you're working on a very large project or one that's been around for a long time, there could be dozens or even hundreds of commits in the log.

We'll cover two of the `git diff` options that can help make our output easier to read. The first is `git log --oneline`:

```
$ git log --oneline
f9658e3 (HEAD -> main) add second day's birds to list
```

```
7cff5a0 add first day's birds to commit
416964f initial commit
```

The `--oneline` option creates an one-line version of the entry for each commit, consisting of the abbreviated SHA and the commit message (or the title line, if it's a multiline commit message).

The second helpful option is `-n` (where `n` represents a number), which just shows the `n` most recent commits. For example, `git log -1` would show the most recent commit, `git log -3` would show the last three, etc. Give it a try!

Check for Understanding

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

1. What is `HEAD` ?
2. What do each of the following commands show: `git diff` , `git diff --staged` , and `git diff HEAD` ?

Conclusion

We've covered a lot of ground in this lesson and learned a lot of new commands. If it feels somewhat overwhelming, that's okay. Remember, the goal here is **not** to memorize a bunch of commands, but rather to get familiar with and practice some of the most useful ones so that you'll be comfortable finding and using the commands you need when you need them.

Resources

- **Git diff Tutorial (https://www.atlassian.com/git/tutorials/saving-changes/git-diff)**
- **Advanced Git Log (https://www.atlassian.com/git/tutorials/git-log)**