To squash commits together, we're going to use the extremely powerful `git rebase` command. This is one of my *favorite* commands, but it did take me *quite* a while to become comfortable with it. At first, it was somewhat challenging for me to get a handle on how it works, and then (after reading countless warnings online) I was scared to actually use it for fear of irreparably damaging my project's Git history.

But I'm here to tell you that `git rebase` isn't really all that difficult, and that you can bravely make changes to your repository without fear of doing any damage! (<-- quite the claim, isn't it!?!)

Let's first get a big picture idea of how squashing works, and then we'll actually do some squashing with the `git rebase` command.

The command I used is:

```
$ git rebase -i HEAD~3
```

The Rebase Command

The `git rebase` command will move commits to have a *new base*. In the command `git rebase -i HEAD~3`, we're telling Git to use `HEAD~3` as the base where all of the other commits (`HEAD~2`, `HEAD~1`, and `HEAD`) will connect to.

The `-i` in the command stands for "interactive". You *can* perform a rebase in a non-interactive mode. While you're learning how to rebase, though, I definitely recommend that you do *interactive* rebasing.

Ancestry References

As a brief refresher, `HEAD` indicates your current location (it could point to several things, but typically it'll either point to a branch name or directly to a commit's SHA). The `~3` part means "three before", so `HEAD~3` will be the commit that's three before the one you're currently on. We're using this relative reference to a commit in the `git rebase` command.

Let me demonstrate how to use this command to combine the three destination commits into one.

QUIZ QUESTION

In the command:

```
$ git rebase -i HEAD~3
```

...the `HEAD~3` is an ancestry reference to a commit that will act as the new base for the commits that are being rebased.

Which of the following could be used as a reference to a base?

- a SHA
- a branch name
- a tag name

SUBMIT

Force Pushing

In the video, I had to force push the branch. I had to do this because GitHub was trying to prevent me from accidentally deleting commits. Because I used the `git rebase` command, I effectively *erased* the three separate commits that recorded my addition of Florida, Paris, and Scotland. I used `git rebase` to combine or *squash* all of these commits into one, single commit.

Using `git rebase` creates a new commit with a new SHA. When I tried using `git push` to send this commit up to GitHub, GitHub knew that accepting the push would erase the three separate commits, so it rejected it. So I had to *force push* the commits through using `git push -f`.

> ***⚠️ Force Pushing ⚠️***

> *In this instance, force pushing my commits was necessary. But if you try to push commits and GitHub rejects them, it's trying to help you, so make sure to review what commits you're pushing and the commits that are on GitHub to verify you're not about to overwrite content on your remote repository accidentally!*

Rebase Commands

Let's take another look at the different commands that you can do with `git rebase`:

- use `p` or `pick` – to keep the commit as is
- use `r` or `reword` – to keep the commit's content but alter the commit message
- use `e` or `edit` – to keep the commit's content but stop before committing so that you can:
  - add new content or files
  - remove content or files
  - alter the content that was going to be committed
- use `s` or `squash` – to combine this commit's changes into the previous commit (the commit above it in the list)
- use `f` or `fixup` – to combine this commit's change into the previous one but drop the commit message
- use `x` or `exec` – to run a shell command
- use `d` or `drop` – to delete the commit

When to rebase

As you've seen, the `git rebase` command is incredibly powerful. It can help you edit commit messages, reorder commits, combine commits, etc. So it truly is a powerhouse of a tool. Now the question becomes "*When* should you rebase?".

Whenever you rebase commits, Git will create a new SHA *for each commit*! This has drastic implications. To Git, the SHA is the identifier for a commit, so a different identifier means it's a different commit, *regardless if the content has changed at all.*

So you should not rebase if you have already pushed the commits you want to rebase. If you're collaborating with other developers, then they might already be working with the commits you've pushed. If you then use `git rebase` to change things around and then force push the commits, then the other developers will now be out of sync with the remote repository. They will have to do some complicated

surgery to their Git repository to get their repo back in a working state...and it might not even be possible for them to do that; they might just have to scrap all of their work and start over with your newly-rebased, force-pushed commits.

Recap

The `git rebase` command is used to do a great many things.

```
# interactive rebase
$ git rebase -i <base>

# interactively rebase the commits to the one that's 3 before the one we're
$ git rebase -i HEAD~3
```

Inside the interactive list of commits, all commits start out as `pick`, but you can swap that out with one of the other commands (`reword`, `edit`, `squash`, `fixup`, `exec`, and `drop`).

I recommend that you create a `backup` branch *before* rebasing, so that it's easy to return to your previous state. If you're happy with the rebase, then you can just delete the `backup` branch!

Further Research

- Git Branching - Rebasing from the Git Book
- git-rebase from the Git Docs
- https://www.atlassian.com/git/tutorials/rewriting-history#git-rebase from the Atlassian blog