#3 Tools for Changing History

#advanced-git

Rebase

There are a couple of good reasons to rebase:

1. **Keeping a branch up to date to it's source branch.** If you're working on a separate development branch, it's very common for master to pull in new changes before you get to merge back in. Updating your work now ensures that your work will seamlessly integrate later.
2. **Searching for opportunities to fast forward.** You won't be allowed to fast-forward if your feature branch lags behind the source branch, and you will definitely have issues doing a fast-forward if there are merge conflicts. Rebasing tackles both of those issues now.
3. **Maintaining a linear commit history.** This ties into the second point above, but many people find it easier to read and understand a linear commit history (free of branching and merges). With merges, there's some extra work that needs to go into understanding how commits on either side of the merge affect each other.

Without further ado, let's talk about how to rebase.

```
git rebase <source_branch>
```

This will rebase the current branch **onto** the source branch.

The rebase isn't "free", however. If there are any conflicts with pushing the base commit up, they will need to be resolved (much like a merge). This won't create any additional commits, though it may end up changing some commit ids.

Amending Commits

Let's say you make a commit, and realize you made a typo in the commit message. Or you forgot to add a missing file. `git commit –amend` will allow you to edit the most recent commit, including editing the commit message.

```
$ git commit
$ git add forgotten.txt # Whoops, forgot a change
$ git commit ––amend # Can also edit commit message
```

Interactive Rebase

The use case for interactive rebase is usually a bit different than "regular" rebase. Interactive rebase allows you to quickly modify many commits, by either removing them, combining them, reordering them, etc. You could technically use it to do a regular rebase, but you might as well use `git rebase` for that.

To do interactive rebase, start with a rebase command and add the `-i` flag. This will do the rebase but give you the opportunity to modify commits as well.

```
$ git rebase master -i
```

If you want to just edit commits without actually changing the base, you can rebase on an earlier point in the branch.

```
$ git rebase HEAD~4 --interactive
```

This will allow you to use commits up to `HEAD~3`

Revert

Revert actually doesn't change history, but it's in this section because it's a very good alternative to modifying the past. Revert will create an "undo commit" of a past commit, which will preserve history but create the net effect of undoing the past.

Reflog

Reflog stands for "Reference Log", which is a kind of metadata for Git. While `git log` will list all the commits created, `git reflog` will list all the *actions* done to modify the repository state. This includes stuff like creating new commits, reset, and even "relatively safe" operations like checkout (when changing `HEAD`).

The following command will show the reflow:

```
$ git reflog
27e7064 (HEAD -> master) HEAD@{0}: commit: Add .gitignore
aeaf99e HEAD@{1}: checkout: moving from weather to master
```

```
16428a9 (weather) HEAD@{2}: checkout: moving from master to weather
aeaf99e HEAD@{3}: merge draft: Fast-forward
0a1221e HEAD@{4}: checkout: moving from weather to master
16428a9 (weather) HEAD@{5}: commit: Add temp
...
```

This shows the location of `HEAD` after all the past commands in your history for this repo. The characters in the left most column represent commit hashes. The text in the right most column shows what command lead to this position of `HEAD`.

For example, `HEAD@{1}` represents the position of `HEAD` one move ago. There are alternative ways of addressing the reflog (such as looking at the position of `HEAD` 1 day ago), which you can find online.