Advanced Homework 2

Due: Wednesday, September 26th, 11:59PM (Hard Deadline)

Submission Instructions

To receive credit for this assignment you will need to stop by someone's office hours, demo your running code, and answer some questions. **Make sure to check the office hour schedule as the real due date is at the last office hours before the date listed above.** This applies to assignments that need to be gone over with a TA only. **Extra credit is given for early turn-ins of advanced exercises. These details can be found on the website under the advanced homework grading policy.**

1 Git Golf

Sometimes, when using git, people make mistakes. Git can be remarkably forgiving, if you've ever told git to remember something, then you can usually find a way to undo your mistake and get it back.

```
# Grab a copy of the files for this question
> wget http://c4cs.github.io/static/f18/advanced/golf.tar.gz
```

#1: Undeleting Files

In this repository, someone cleaning up ran `git rm *.png`, which deleted every image, and committed the result. Now the website is broken because an image that should have stayed was deleted. While you could use `git revert` to undo the commit, it also changed the website source, so you really don't want to undo everything.

Demonstrate how to recover the missing picture without reverting the commit that deleted it.

#2: Undeleting Commits

Sometimes it is possible to lose a commit. This can happen because you deleted a branch, a rebase went poorly, or a reset went awry. Regardless of how it happened, there are ways of finding commits that nothing is currently pointing to. This repository has such a commit.

Demonstrate how to recover a commit that nothing is currently pointing to.

#3: Undeleting Changes

When working with git, `git add my_file` stages a file, but it isn't actually committed until you run `git commit`. Sometimes you change your mind after a `git add` and run `git reset my_file` to unstage a file. The changes to that file are still there, however. To really undo changes, you use `git reset --hard`.

Once you've been using these commands for a little while, it can be a little too easy to accidentally type the wrong thing. In this repository, someone accidentally typed `git reset --hard` when they meant to just type `git reset`. Fortunately, because they had already run `git add` to stage their changes, the deleted changes can be recovered.

Demonstrate how to recover changes that had been staged for commit, but were then deleted.

Submission checkoff:

☐ Explain how you solved each problem

2 Automating Professionalism

One feature of most version control systems is *hooks*. A hook is an automated script or tool that runs at various points in time. For example, later in this course we will show how you can automatically run test cases every time anyone commits code.

A slightly easier one to wire up, however, is a hook that will automatically check the spelling of your commit messages for you, letting you know if you made any mistakes. While spelling rarely counts for a class projects, it adds a nice bit of professionalism to any future work you'll share with others.

Like the regular homework, I recommend creating a temporary git repository to play with while you test things and try to get things working.

Git stores all of its information and configuration in a folder named `.git` in the root of each project. Navigate to `.git/hooks` and rename `commit-msg.sample` to `commit-msg`. This activates the commit message hook, which runs after you write and save your commit message. Now go back to your repository, make a change, and commit it. Hmm, doesn't look like anything changed. Make another change, but this time make this your commit message (**exactly this, capitalization matters**):

```
This is a test.

Signed-off-by: Me!
Signed-off-by: Me!
```

After trying to commit, type `git status`. What happened? Go back and look at the commit hook. Do you understand what it does? Try running `git commit --no-verify` with the same commit message. Go edit the commit hook and delete the line `exit 1`. Try making a new commit with the same commit message, what happens now? What is `$1` in this script? (not sure? try adding lines like `echo $1` or `echo $(file $1)` or `echo $(cat $1)` to the hook and making commits, what happens?)

`aspell` is a simple command-line utility that checks spelling. Install it and play with it a little.

Write a commit hook that checks the spelling of a commit message. Your hook should not prevent the commit from going through (that'd be annoying..), but it should print out any spelling errors. Your hook should also print a message that suggests running the command `git commit --amend`[1] if any errors are present.

Some tips

- This is a little intimidating to get started. *Try some things.* Make a bunch of garbage commits, modify the hook a little, see what happens.
- A shell script is exactly like working in a terminal. The only magic is that you've typed all the commands in advance instead of one at a time. So try some things in your terminal! Mess around until you get some commands that do what you want, then copy them to your script.
- Shell scripting is hard and ugly though. Maybe write a Python script to help you out and call it? Then again, calling commands from Python is kinda hard, so maybe not. *Whatever you are more comfortable with.*
- There is **zero** need to be efficient. This hook is called rarely and operates on hundreds of bytes of text. Read the file 6 times. Write 7 temporary files. *Who cares.* The goal is not to be pretty, the goal is to work.

[1] Amending a commit lets you change your most recent commit. This is fine when you are working locally, but can be a dangerous command if you are sharing your repository with anyone else. Use with caution.

- Speaking of temporary files, the `/tmp` directory can be a great place to throw those. There's even a command called `mktemp` that will give you a new, unique filename. (You can also do this without making any temporary files, it's just a little harder)

Submission checkoff:

☐ Show off your spellchecking hook in action, explain how it works

Optional **Extra Credit (1 pt)**

☐ Demonstrate that your hook ignores errors in indented lines, explain how it works

Further exploration and some gotchas:

- Take a look at some of the other hooks, are any of them useful?
- For a list of all available hooks, type `man githooks`.
- Hooks have to be marked as executable to run (`chmod +x`). The sample hooks already had the executable bit set, which is why renaming the existing sample worked above.
- Hooks can call other scripts. Because invocation of hooks is controlled by the name of the script, if you want multiple scripts to run for a single hook, you'll need to have one script named correctly that calls your other hooks.