Practical Git

Note: these examples are designed to **defeat** git's automatic behaviour and cause odd behaviour and conflicts, so we can learn how to fix it. All contain code in both C and Fortran. Use whichever you prefer

- Before you start

Since we cannot have git repositories inside git repositories, these exercises come in the form of simple bash scripts which create the necessary histories. Follow the instructions on Github to get set up.

- Use Directory "One"

- Demonstrates a simple ambiguous merge

Info:

This repo. has 2 branches, master and feature containing slightly inconsistent code.

Try:

Have a look at the code on master and feature separately. What's different?

The command `git log --all --decorate --oneline --graph` will give a 'pretty' graph of the history. Note how things have diverged

Checkout master and try to merge the branch 'feature'. What happens? Why?

Fix the conflict, however you wish, add the file and commit to finish the merge

Extra: create a new commit adding the 'F' case to master. Does this now merge cleanly onto feature? If you do this, what happens to the history?

- Use Directory "Two"

- Demonstrates feature collision that git handles badly

Info:

This repo. has 2 branches, master and feature, containing separate implementations of the same feature

Try:

Have a look at the code on master and feature separately. What's different?

You can compare a single file across branches using `git diff feature master -- eg.f90` (Note the '--' denoting the end of the args) Make sure the diff agrees with your assessment of what is different.

Checkout master and try to merge the branch 'feature'. How bad is the result?

Fix the conflict, however you wish. Run the code, make sure it compiles and ensure it reproduces the previous result.

Extra: does it help to improve the surrounding context by moving the set and print to the same place in both branches? Note - first undo the merge you made before using `git reset <commit just before merge>`

- Use Directory "Three"

- Demonstrates git's use of context for matching code blocks

Info:

This repo. has 2 branches, master and feature containing code with a few differences.

Try:

Have a look at the code on master and feature separately. What's different?

Why doesn't the diff work very well?

Try using -b to ignore changes to whitespace

Try using --word-diff=color to generate the diff by "words" (split by white space)

Note: these just make for prettier diffs that are easier to interpret. They don't affect what gets committed

- Use Directory "Four"

- Demonstrates that reverting means removing from history

Info:

This repo. has 3 branches, master, feature1 and feature2. Suppose we added something to master, and then created feature1 and feature2. We realise we didn't want that last master commit yet on feature1, so we revert it and carry on. On feature2, we keep working. Later, we forget what we've done and merge feature1 and feature2 into master…

Try:

Merge feature1 into master. Compile and run. Merge feature 2. Fix the obvious conflict. Why is there a conflict with nothing? Does the code now compile? What happened?

Revert the revert (yes, this is the 'proper' way to do it). To do this you need to find the ID of the revert. Note the new conflict.

Extra: assuming you only want part of a reverted commit back, how could you get just the changes you want? (There are several right answers to this.)

- Use Directory "Five"

  - Demonstrates a merge that does the wrong thing

Info:

This repo. has 2 branches, master and feature. Since we created the feature branch, other things were added to master. We now have to submit our feature for inclusion.

Try:

Compile and run the code on branch 'feature'. Merge master into feature. Fix conflicts. Compile. Run. What has gone wrong?

- Final Test

  - Demonstrates a real merge

Info:

This repo. has 3 branches, master and bill/cleanup and ben/cleanup. Bill and Ben made different changes to clean up this code. We are tasked with merging this.

Try:

Compile and run the code on both cleanup branches. The results should match. Merge both cleanup branches into master, one at a time. Fix the conflicts. Congratulations.

- Lessons to take away

- We deliberately broke git's ability to track changes a bit a time here. This isn't hard to do and can have unexpected effects.
- The trick is not to blindly trust merges. They can go wrong, so always compile and run the result, and if possible run tests on it.
- Even when merges work OK, we can end up re-doing a lot of changes if two people edit the same lines of code
- Finally, it does help to regularly merge master into something you're working on so you aren't faced with merge-horror at the end
  - Note though that this can mean your history ends up full of commits that **don't work.** Sometimes it is better to merge a feature in in completed form instead.