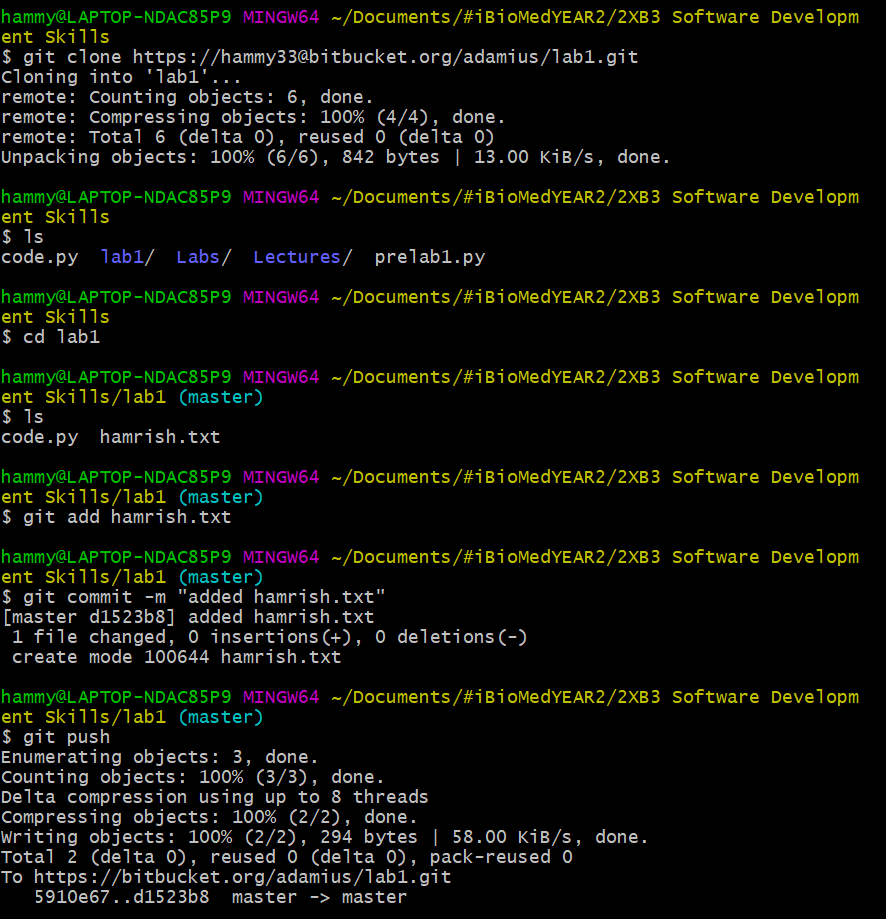
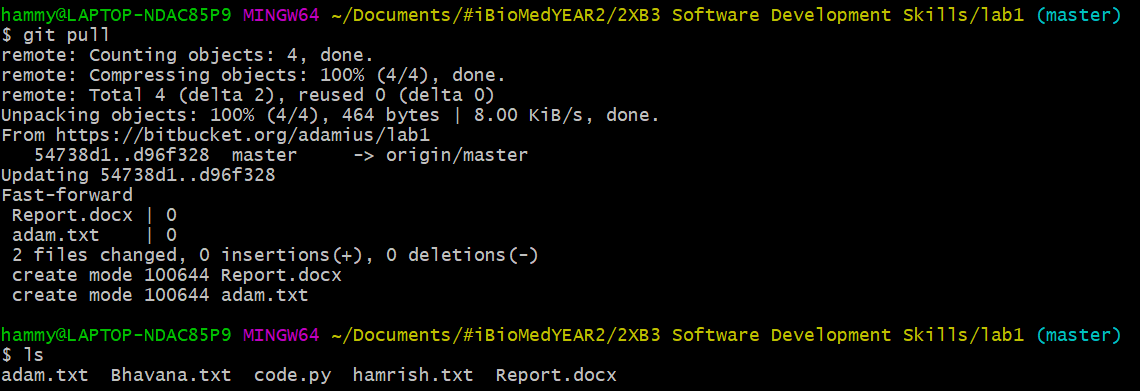
**Git Push & Git Pull**

*git push:* Used to upload contents of a local repository to a remote repository

*git pull:* Used to fetch contents from a remote repository in order to update the local repository

Proof of adding, committing, and pushing “hamrish.txt” to the remote repository, and pulling from remote repository to update the local repository.

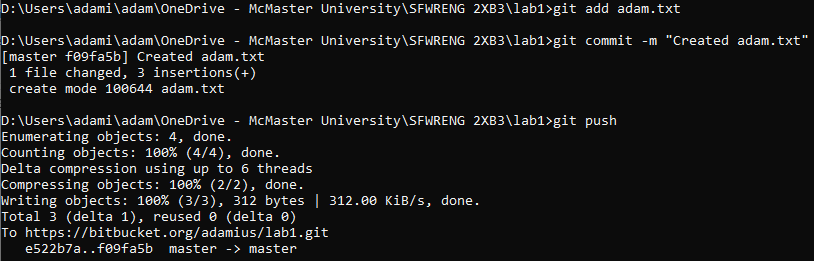
 

Proof of adding, committing, and pushing “Bhavana.txt” to the remote repository.

Text

Description automatically generated

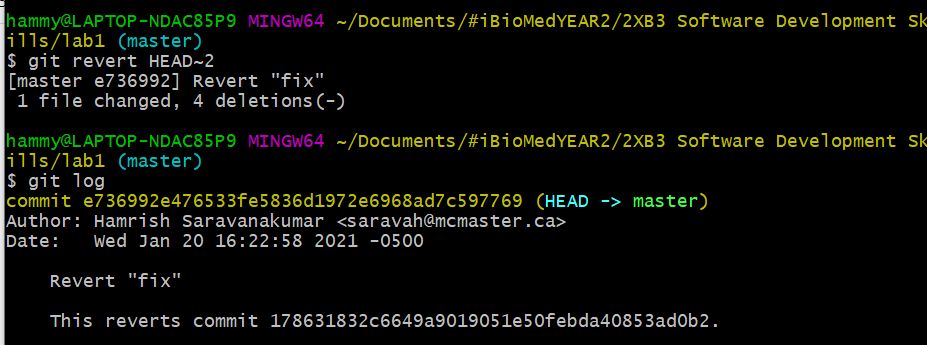
Proof of adding, committing, and pushing “adam.txt” to the remote repository.



**Git revert & Git reset**

***git revert****:* This command serves as a method of undoing changes, but instead of completely deleting commits in the commit history, the revert will create an entirely new commit that inverse any particular change specified

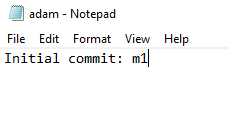
1. Use *git log* to view past commits
2. Decide which commit you’d like to undo until. Use one of the following two commands:
   1. git revert HEAD~<n>
      1. n = the number of commits you want to revert
   2. git revert <SHA or hash>
      1. If you want to go back multiple commits, you must include the SHA1 of every commit, separated by a space
3. Use *git log* again to view the reverted commit.

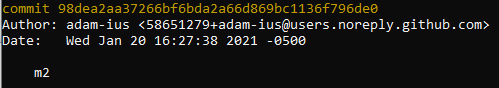
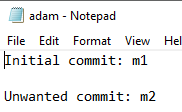


The image above shows a successful use of git revert by undoing two past commits, as shown in when using git log to view the commit history.

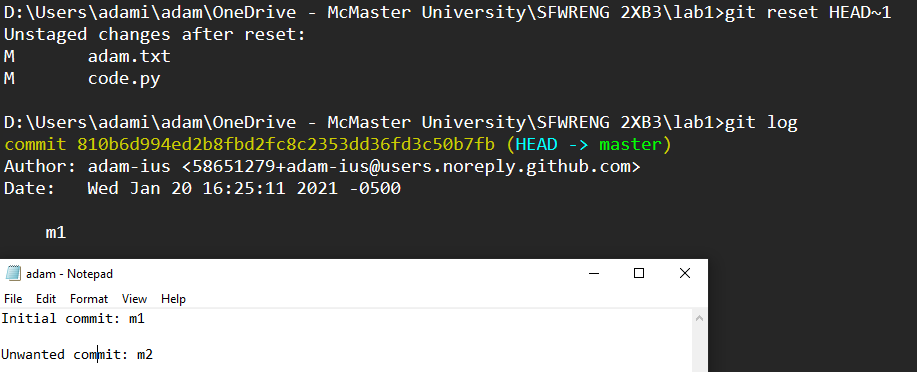
***git reset:*** This command is used to undo local changes to the current state of a git repository. The pointer that points to the current branch with the most recent commit, and points this elsewhere. Using *git reset* should be used on a private branch, where the current commit hasn’t been pushed.

1. Use *git log* to view past commits
2. Decide which commit you’d like to undo until. Use one of the following two commands:
   1. git reset HEAD~<n>
      1. n = the number of commits you want to revert
   2. git reset --hard HEAD~<n>
3. Use *git log* again to check that HEAD points to the right branch.
4. If you did not use the –hard flag, you will have to *git checkout* <SHA or hash> -- <file> to update the rolled back contents of the file locally.

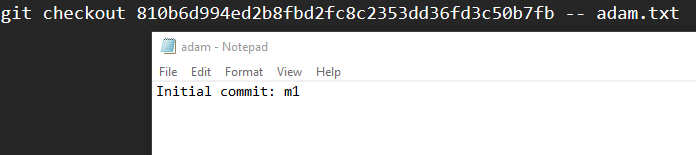


Initial commit. (m1)

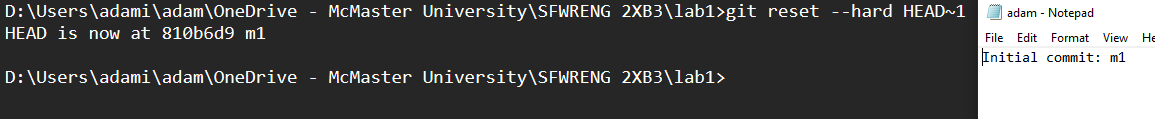
Unwanted commit. (m2)



*git reset* HEAD~1 moves the HEAD pointer back 1 space (from m2 to m1). However, the contents of adam.txt still show the unwanted commit contents.



*git checkout* (m1 hash) -- adam.txt updates the contents to the correct commit (m1).



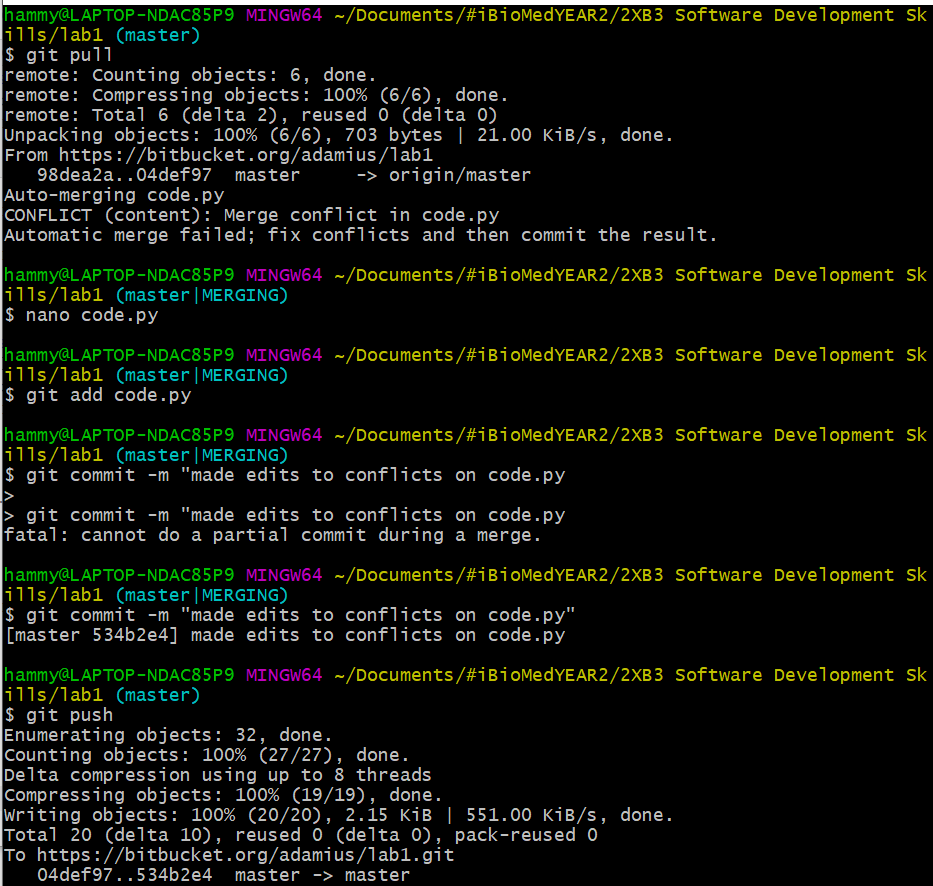
*git reset --hard* HEAD~1 moves the HEAD pointer back 1 space **and** changes the contents of all files to reflect the correct branch (m1).

If a commit has been made somewhere in the repository history, and one later decides that the commit is wrong and should not have been done, then git revert allows for someone to undo the changes introduced by the bad commit, recording the "undo" in the history. It acts as a nicer way to cancel out changes because the Git workflow makes it challenging if a set of commits is no longer seen in the log. However, if you have made a commit, but haven't actually shared it with anyone else and you are confident that it isn’t needed, then git reset would be a better option in order to rewrite the history so that it looks as though you never made that particular commit. When working with a group of people,

**Initial Start to code.py**

1. One member created an empty code.py file and pushed it to the remote repository.
2. All other members pulled to each of their local repositories.
3. All 3 members edited the code.py file to implement the are\_valid\_groups function
4. One group member was successfully able to push their changes without any issues, however the other 2 group members were forced to pull from the repository first, and edit the merge conflicts that arose from having a different file from the most recent push.
5. Once merge conflicts were handled manually, every group member did a final pull to ensure each local repository was up to date.

Proof of pushing changes and resolving conflicts on code.py.



**Players & Adversaries**

*Steps Taken*

1. Adversary #1 made minor edits to “annoy” the player, and added, committed and pushed changes to code.py first.
2. Adversary #2 made minor edits to “annoy” the player. This person pulled in order to merge the changes with the initially code.py file and the code.py file that was edited by adversary #1. Once merge conflicts were resolved manually, adversary #2 added, committed and pushed changes to code.py.
3. Player implemented the changes to code.py as requested in the lab documentation. Prior to implementing such changes, the player pulled in order to merge the changes with initial code.py file with the code.py file that was edited by adversary #1 and adversary #2. Once merge conflicts were resolved manually, the player added, committed and pushed the changes to code.py.
4. The player accessed the commit history (using git log), and used git reset in order to the state of the file prior to when the player was selected.
5. Process was repeated until all 3 group members served as the player.

**Rotation 1:**

Text

Description automatically generated A picture containing text

Description automatically generated

Code.py following merge conflict

Graphical user interface, text, application

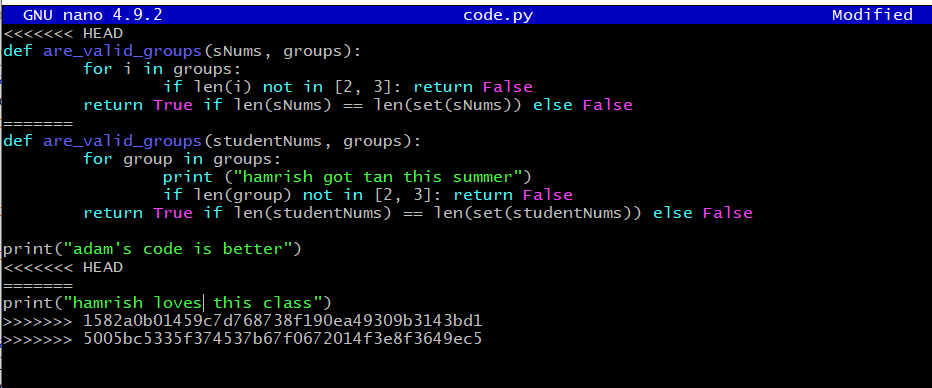
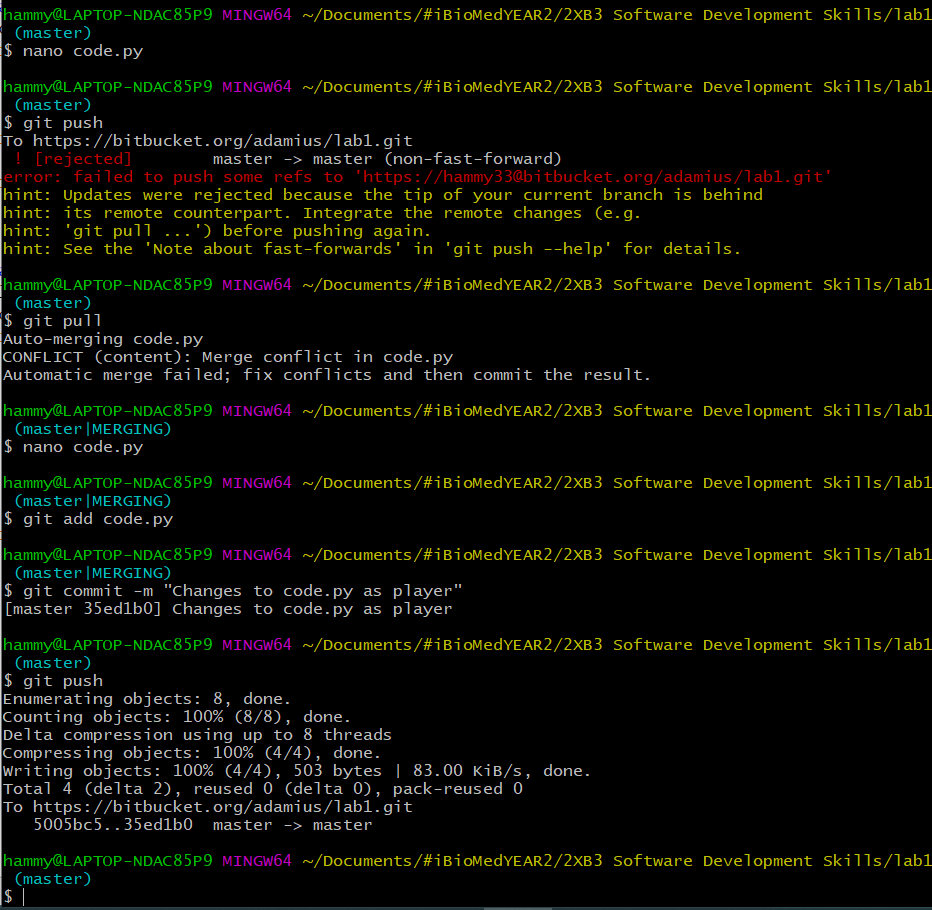
Description automatically generated

Code.py after manually resolving conflict

**Rotation 2:**



**Rotation 3:**

**Reflection on Conflict**

Through this lab, we were exposed to the idea of a merge conflicts, as well as how and why one might take place. We often experienced situations where we needed to merge incoming changes from the remote branch to our local branch. Any time a file that we were working on was modified by one of our partners and pushed to the local repository, we were forced to resolve the conflict before successfully pushing our changes to that particular file. Most of the time, going into the file and making manual edits to resolve the conflict served as the easiest way, but would take time. Upon experiencing this issues with conflicts, there were multiple ways in which we learned to resolve these issues throughout the lab

There were often times where we thought we had resolved the merge, but were immediately faced with an error messages such as "unable to merge" or "you have unmerged paths" which made the process quite frustrating. We learnt to use *git status* to view the current state of the branch to ensure that all changes had been merged successfully. Using *git merge* was also seen as helpful, as this allowed us to start over and return to a clean slate in order to attempt to resolve the conflict. Getting exposed to *git revert* additionally allowed us to undo particular commits that were undesirable, while still having an accurate history to look back on.