Git Exercises

Link to repo: https://github.com/lynn511/MLOps_usj

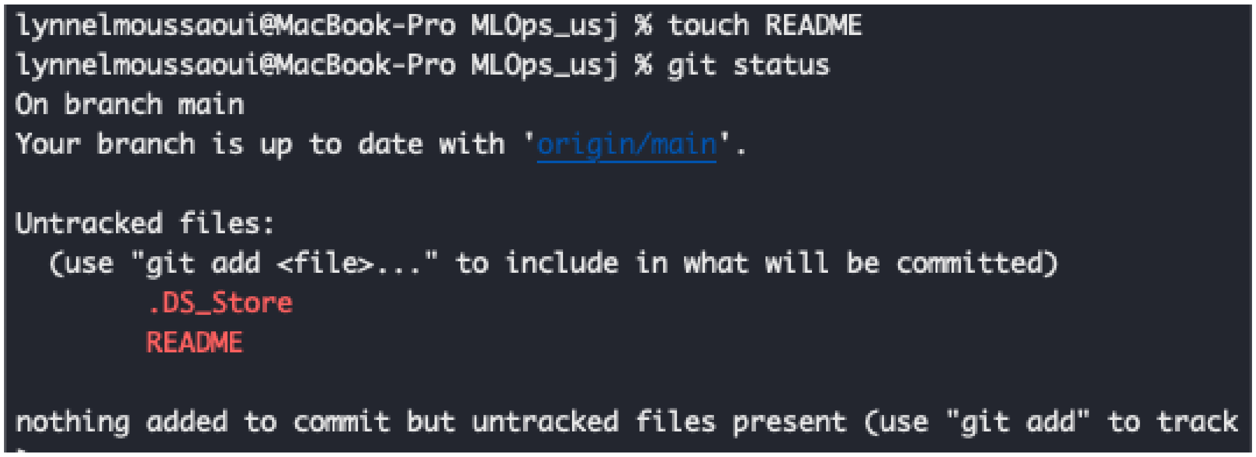
Introduction

These exercises aim to give you some practice with using the Git version control system. Each exercise comes in two parts: a main task that most, if not all, course attendees should be able to complete in the allocated time, as well as a stretch task for those who complete the main task quickly.

Exercise 1 - Tracking Files

Main Task

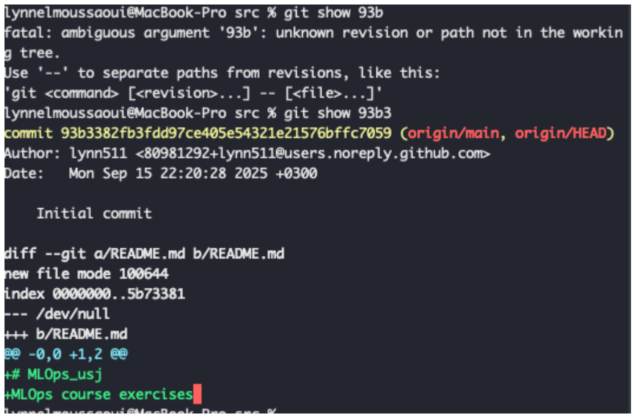
1. Create a new directory and change into it.
2. Use the **init** command to create a Git repository in that directory.
3. Observe that there is now a **.git** directory.
    a. What is it used for ?
    It's a hidden file, it contains git's internal data
4. Create a **README** file.
5. Look at the output of the **status** command; the **README** you created should appear as an untracked file.



```
lynnelmoussaoui@MacBook-Pro MLOps_usj % touch README
lynnelmoussaoui@MacBook-Pro MLOps_usj % git status
On branch main
Your branch is up to date with 'origin/main'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .DS_Store
        README

nothing added to commit but untracked files present (use "git add" to track
```

6. Use the **add** command to add the new file to the staging area. Again, look at the output of the **status** command.
    a. In which stage does the file appear ? it is in the staging area
7. Now use the **commit** command to commit the contents of the staging area.
8. Create a **src** directory and add to it two new empty files: *file1.py* and *file2.py*.
9. Use the **add** command on the directory, not the individual files. Use the **status** command. See how both files have been staged, then Commit them.

10. Make a change to *file1.py*. Use the **diff** command to view the details of the change.
11. Next, **add** the changed file, and notice how it moves to the staging area in the **status** output.
    a. Observe that the **diff** command you did before using **add** now gives no output.
    b. Why not? What do you have to do to see a **diff** of the things in the staging area? Diff shows us unstaged changes, after we did git add there's no unstaged changes
12. Without committing, make another change to the same file you changed in step 10. Look at the **status** output, and the **diff** output.
    a. Notice how you can have both staged and unstaged changes, even when you're talking about a single file.
    b. Observe the difference when you use the **add** command to stage the latest round of changes.
    c. Finally, **commit** them. You should now have started to get a feel for the staging area.
13. Use the **log** command in order to see all of the commits you made so far.
14. Use the **show** command to look at an individual commit.
    a. How many characters of the commit identifier can you get away with typing at a minimum? I tried and it worked with 4 characters minimum



```
lynnelmoussaoui@MacBook-Pro src % git show 93b
fatal: ambiguous argument '93b': unknown revision or path not in the workin
g tree.
Use '--' to separate paths from revisions, like this:
'git <command> [<revision>...] -- [<file>...]'
lynnelmoussaoui@MacBook-Pro src % git show 93b3
commit 93b3382fb3fdd97ce405e54321e21576bffc7059 (origin/main, origin/HEAD)
Author: lynn511 <80981292+lynn511@users.noreply.github.com>
Date:   Mon Sep 15 22:20:28 2025 +0300

    Initial commit

diff --git a/README.md b/README.md
new file mode 100644
index 0000000..5b73381
--- /dev/null
+++ b/README.md
@@ -0,0 +1,2 @@
+# MLOps_usj
+MLOps course exercises
```

15. Make a couple more commits, at least one of which should add an extra file.

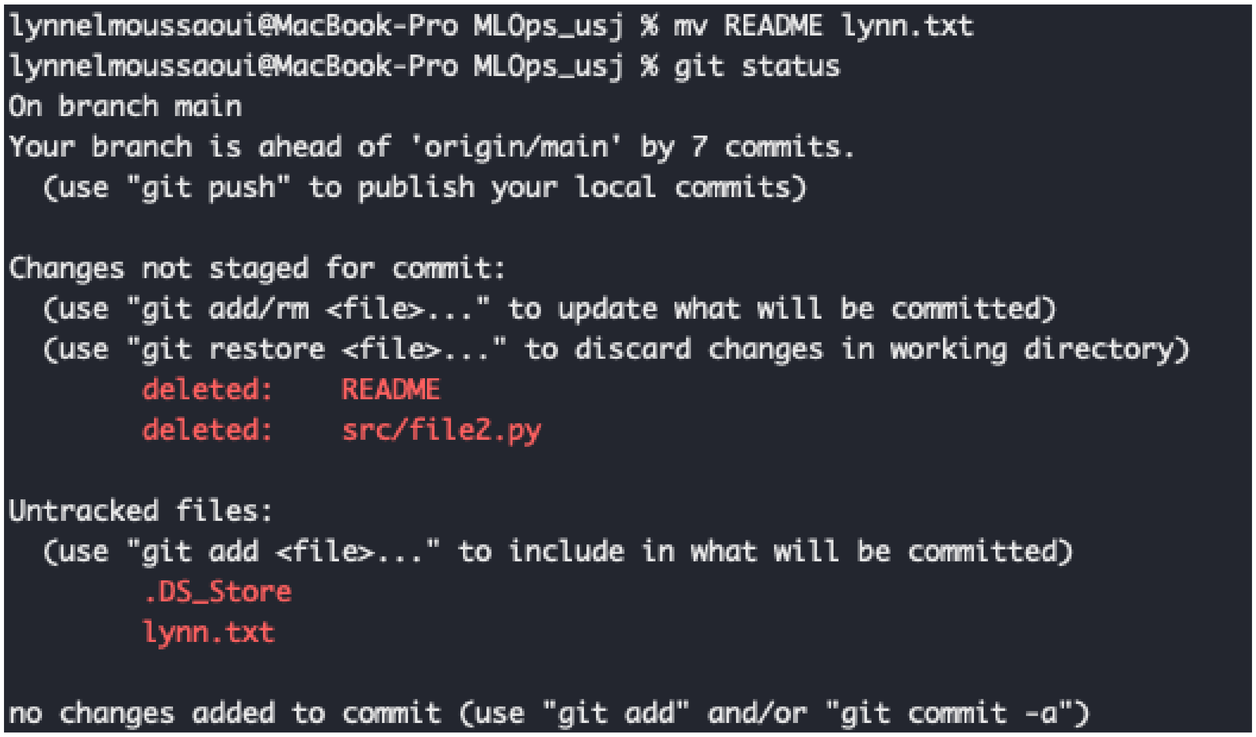
Stretch Task

1. Use the Git **rm** command to remove a file. Look at the **status** afterwards. Now **commit** the deletion.
2. Delete another file, but this time do not use Git to do it; e.g. if you are on Linux, just use the normal (non-Git) **rm** command; on Windows use **del**.
3. Look at the **status**. Compare it to the status output you had after using the Git built-in **rm** command. Is anything different? After this, **commit** the deletion.
   When used git rm, it is under changes to be committed. But by using built in rm the change is not staged for commit, I should still do git add to stage the change, since it happened in the working directory
4. Use the Git **mv** command to move or rename a file; for example, rename **README** to **README.md**. Look at the status, then commit the change.

Exercise

Main

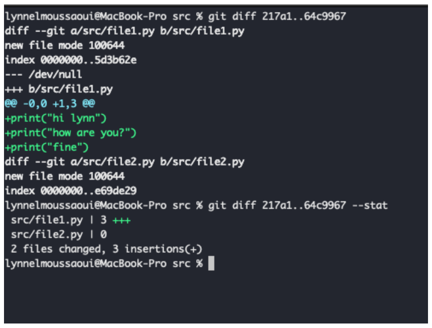
5. Now do another rename, but this time using the operating system's command to do so. (same as question 2) How does the status look?



```
lynnelmoussaoui@MacBook-Pro MLOps_usj % mv README lynn.txt
lynnelmoussaoui@MacBook-Pro MLOps_usj % git status
On branch main
Your branch is ahead of 'origin/main' by 7 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    README
        deleted:    src/file2.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .DS_Store
        lynn.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

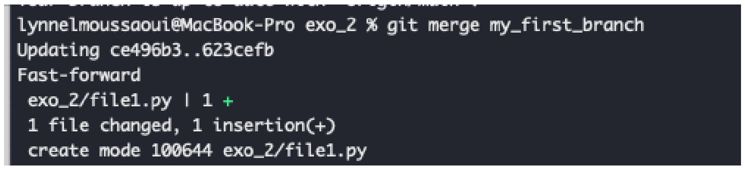
   a. Will you get the right outcome if you were to **commit** at this point? No,it did not track a rename
   b. Work out how to get the **status** to show that it will not lose the file, and then commit. I did git add lynn.txt README, it shows as renamed
   c. Did Git at any point work out that you had done a rename? After staging

6. Use git help log to find out how to get Git to display just the most recent 3 commits.
7. Try using **--stat** option with **show** command. Test it with **log** and **diff** commands.
   a. What does it do ? summarizes changes
8. Imagine you want to see a diff that summarises all that happened between two commit identifiers. You can use the **diff** command, specifying two commit identifiers joined by two dots (that is, something like **abc123..def456**). Check the output is what you expect.



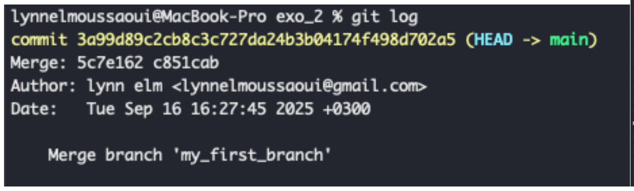
```
lynnelmoussaoui@MacBook-Pro src % git diff 217a1..64c9967
diff --git a/src/file1.py b/src/file1.py
new file mode 100644
index 0000000..5d3b62e
--- /dev/null
+++ b/src/file1.py
@@ -0,0 +1,3 @@
+print("hi lynn")
+print("how are you?")
+print("fine")
diff --git a/src/file2.py b/src/file2.py
new file mode 100644
index 0000000..e69de29
lynnelmoussaoui@MacBook-Pro src % git diff 217a1..64c9967 --stat
 src/file1.py | 3 +++
 src/file2.py | 0
 2 files changed, 3 insertions(+)
lynnelmoussaoui@MacBook-Pro src % 
```

Task - Git Branches

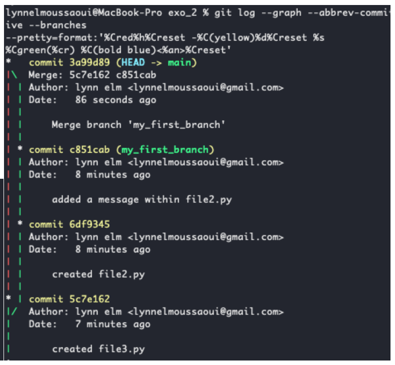
1. Run the **status** command. Notice how it tells you what branch you are in.
2. Use the **branch** command to create a new branch named *my_first_branch*.
3. Use the **checkout** command to switch to it.
4. Make a couple of commits in the branch – perhaps adding a new file and/or editing existing ones.
5. Use the **log** command to see the latest commits. The two you just made should be at the top of the list.
6. Use the **checkout** command to switch back to the master/main branch. Run **log** again.
    a. Notice your commits don't show up now.
    b. Check the files also – they should have their original contents.
7. Use the **checkout** command to switch back to your branch.
    a. Use **log --graph** to take a look at the commit graph; notice it's linear.
    b. You can use this command for a prettier format:
    ```
    git log --graph --abbrev-commit --date=relative --branches --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset'
    ```
8. Now **checkout** the master/main branch again. Use the **merge** command to merge your branch into it.
    a. Look for information about it having been a fast-forward merge.
    We can see fast forward written
    
    ```
    lynnelmoussaoui@MacBook-Pro exo_2 % git merge my_first_branch
    Updating ce496b3..623cefb
    Fast-forward
     exo_2/file1.py | 1 +
     1 file changed, 1 insertion(+)
     create mode 100644 exo_2/file1.py
    ```
    b. Look at the git log, and see that there is no merge commit.
    c. Take a look at the commit graph and see how it is linear.
9. Switch back to your branch (*my_first_branch)*. Make a couple more commits.
10. Switch back to master/main. Make a **commit** there, which should edit a different file from the ones you touched in your branch, to ensure there will be no conflict.
11. Now **merge** your branch again.
12. Look at **git log**. Notice that there is a merge commit. Also look at the commit graph using command from question 7. Notice the DAG now shows how things forked, and then were joined up again by a merge commit.

Exercise

Main



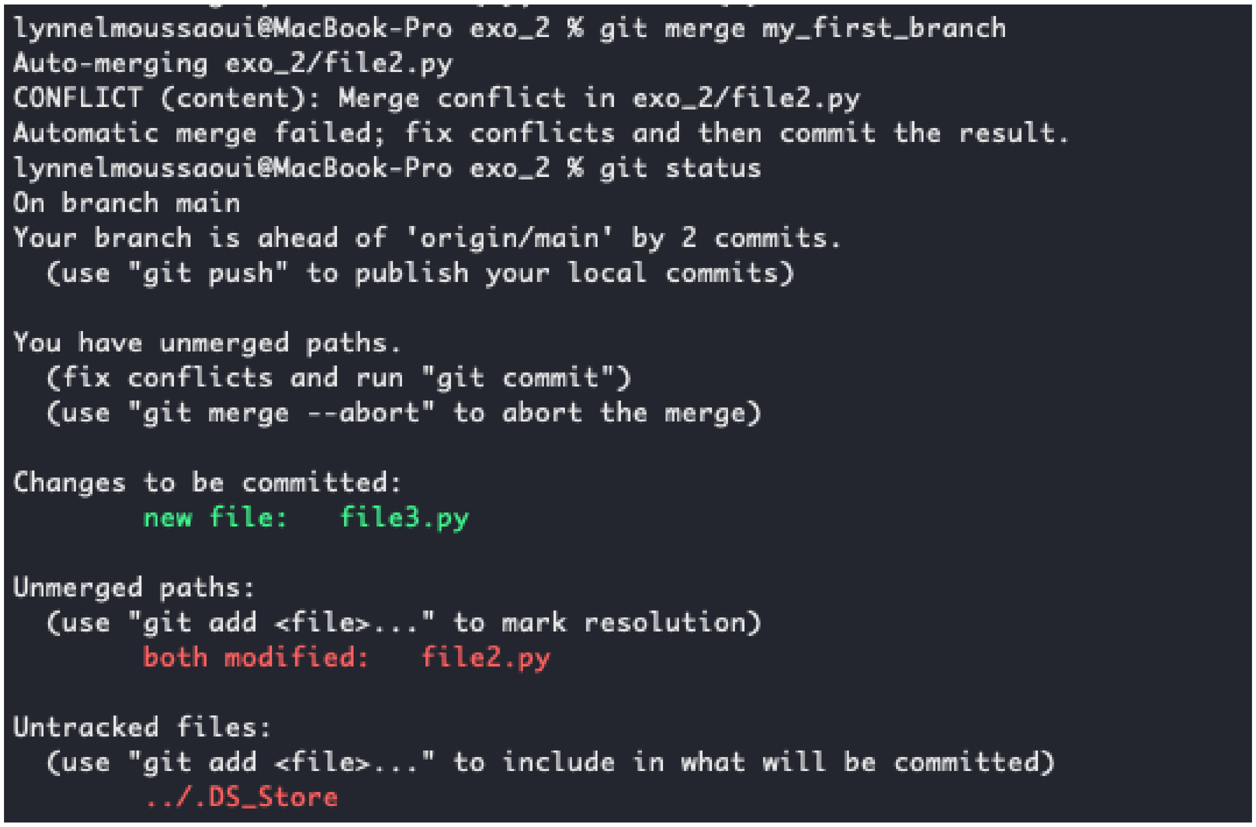
```
lynnelmoussaoui@MacBook-Pro exo_2 % git log
commit 3a99d89c2cb8c3c727da24b3b04174f498d702a5 (HEAD -> main)
Merge: 5c7e162 c851cab
Author: lynn elm <lynnelmoussaoui@gmail.com>
Date:   Tue Sep 16 16:27:45 2025 +0300

    Merge branch 'my_first_branch'
```



```
lynnelmoussaoui@MacBook-Pro exo_2 % git log --graph --abbrev-commi
ive --branches
--pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s
%Cgreen(%cr) %C(bold blue)<%an>%Creset'
*   commit 3a99d89 (HEAD -> main)
|\  Merge: 5c7e162 c851cab
| | Author: lynn elm <lynnelmoussaoui@gmail.com>
| | Date:   86 seconds ago
| |
| |     Merge branch 'my_first_branch'
| |
| * commit c851cab (my_first_branch)
| | Author: lynn elm <lynnelmoussaoui@gmail.com>
| | Date:   8 minutes ago
| |
| |     added a message within file2.py
| |
| * commit 6df9345
| | Author: lynn elm <lynnelmoussaoui@gmail.com>
| | Date:   8 minutes ago
| |
| |     created file2.py
| |
* | commit 5c7e162
|/  Author: lynn elm <lynnelmoussaoui@gmail.com>
|   Date:   7 minutes ago
|
|       created file3.py
```

Stretch Task

1. Once again, **checkout** your branch (*my_first_branch)* and make a couple of commits.
2. Return to your master branch. Make a commit there that changes the exact same line, or lines, as commits in your branch did.
3. Now try to **merge** your branch. You should get a merge conflict.
4. Use git **status**. What do you see ?



```
lynnelmoussaoui@MacBook-Pro exo_2 % git merge my_first_branch
Auto-merging exo_2/file2.py
CONFLICT (content): Merge conflict in exo_2/file2.py
Automatic merge failed; fix conflicts and then commit the result.
lynnelmoussaoui@MacBook-Pro exo_2 % git status
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Changes to be committed:
        new file:   file3.py

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   file2.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ../.DS_Store
```

5. To resolve the conflict:
   a. Open the file(s) that is in conflict and Search for the conflict marker.
   b. Edit the file to remove the markers and choose which code to keep.
   c. Save and quit.

6. Now try to **commit**.
    a. Does it work ? No
    b. Notice that Git still thinks that there are conflicts to resolve.
    c. Look at the output of **status** to understand what is happening.
    There exists unmerged files
7. Use the **add** command to add the files that you have resolved conflicts in to the staging area. Then use **commit** (without a message) to commit the merge commit.
8. Take a look at **git log** and **git log --graph**, and make sure things are as you expected.
9. If time allows, you may wish to...
    a. Delete everything but your **.git** directory, then do a **checkout** command. Just proving that this really will restore all of your current working copy.
    b. Create a situation where one branch has changed a file, but the other branch has deleted it. What happens when you try to merge? How will you resolve it?
    I personally chose to keep the file, I deleted the markers within the file that is doing the conflict then add then commit
    Another solution would be to delete the file as a whole :)
    c. Look at the help page for merge, and find out how you specify a custom message for the merge commit if it is automatically generated.
    d. Look at the help page for merge, and find out how to prevent Git from automatically committing the merge commit it generates, but instead give you a chance to inspect it and merge it yourself. --no-commit

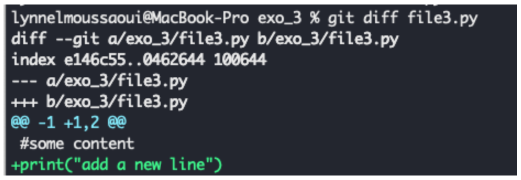
Exercise

Main

3 : Undoing Changes in git

Task: Undoing Changes in Git

1. From your main/master branch, create a new branch called ‘*undoing_changes*’
2. Use git checkout to change into that new branch.
3. Create a new file called `file3.py`, write some content into it, and add it to the staging area.
4. Use the `commit` command to commit the file.
5. Edit the contents of `file3.py` by adding a new line.
    a. Use the `diff` command to see the changes.



```
lynnelmoussaoui@MacBook-Pro exo_3 % git diff file3.py
diff --git a/exo_3/file3.py b/exo_3/file3.py
index e146c55..0462644 100644
--- a/exo_3/file3.py
+++ b/exo_3/file3.py
@@ -1 +1,2 @@
 #some content
+print("add a new line")
```

6. Use the `git checkout` command to discard the changes to `file3.py`.
    a. Use `status` to verify the changes have been undone.
    b. What happened to the changes you made in step 5? They are gone
    c. Could you have achieved the same result with a different command ? Hint: Check the message of `git status` `using the restore command`
7. Create and commit a new file called `file4.py`.
8. Use the `git revert` command to undo the commit you just made.
    a. What does the `revert` command do compared to `checkout`?

       It is on a different scope, revert is on commited directory the commit still exists but the file is deleted, checkout is on the local directory log has no history of what happened
    b. Check the commit history using the `log` command. The commit still exists
    c. What do you notice about the new commit created by `git revert`?

       Hint: You can use `git show HEAD` to see the changes of the last commit.
9. Make a new commit with changes in both `file3.py` and `file4.py`.
10. Use `git reset --soft HEAD^`
    a. Use the status command to check the changes.
    b. Can you describe what happened ? last commit disappeared
    c. Do you still see your previous commit ? no
    d. What could you have done to avoid losing the commit after doing a reset ? git revert
11. Now, use `git reset HEAD`

   a. Check the status again. What is the difference compared to the previous step ? my file is unstaged, it moved my file from staging to working dir
   b. Notice how we use HEAD instead of HEAD^ now ? Why are we doing that ? HEAD^ refers to the commit before the current one, it moves history back. A plain HEAD refers to current commit does not move history
   c. Notice also that we didn't pass any scope to our reset; Git is using the default scope for that command. What do you think that scope is ? the default one is mixed reset
12. Finally, use `git reset --hard`
   a. What happens to both the working directory and the commit history? The changes in the file are gone, the commit log is deleted
   b. We didn't use any commit for this command, what do you think git used as default value ? the HEAD
   c. What would have happened if you had done this command directly after step 9 ?

      Wouldve lost the commits changes as a whole

13. Use the `log` command to verify the commit history after all resets and reverts.

14. Make a couple more commits, at least one of which should add a new file `file5.py`.

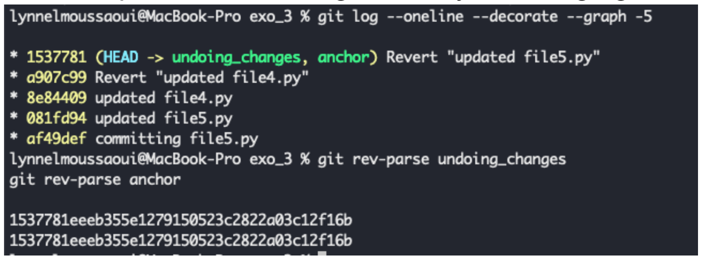
Stretch Task

1. Make some changes to a tracked file (e.g. `file5.py`),
2. Use `git checkout -- <filename>` to undo the changes in that file.
   a. What happens to the file after using `checkout`? Modifications in the file are gone
   b. What does it do to the working directory? File looks like the last commit, my edit does not exist
   c. What happens to the HEAD pointer ? nothing
3. Make two new commits, each changing different files.
4. Then, use `git revert HEAD~2..HEAD`.
   a. What happens to the commit history? Two new commits created
   b. How are the changes handled? The added texts within the files are gone
   c. Can you figure out what this syntax means ? `HEAD~2..HEAD` selects last two commits
   d. Do you think we can use this same syntax with `git reset` ? no in reset we can pick only one point to reset to
5. Use git log --oneline to check your commit history.
6. Create a branch named 'anchor' on your current commit to avoid losing your history.
7. Use git reset HEAD~3
   a. What happened ? 3 last commits gone

Exercise

Main

b. Could you figure out a way to move back your current branch to 'anchor' ? Make sure to update the working directory and staging area accordingly.



```
lynnelmoussaoui@MacBook-Pro exo_3 % git log --oneline --decorate --graph -5

* 1537781 (HEAD -> undoing_changes, anchor) Revert "updated file5.py"
* a907c99 Revert "updated file4.py"
* 8e84409 updated file4.py
* 081fd94 updated file5.py
* af49def committing file5.py
lynnelmoussaoui@MacBook-Pro exo_3 % git rev-parse undoing_changes
git rev-parse anchor

1537781eeeb355e1279150523c2822a03c12f16b
1537781eeeb355e1279150523c2822a03c12f16b
```

c. Test out the different scopes of reset from anchor. You can use the previous step to iterate and test them on the same commits.

i Notes: soft keeps changes staged

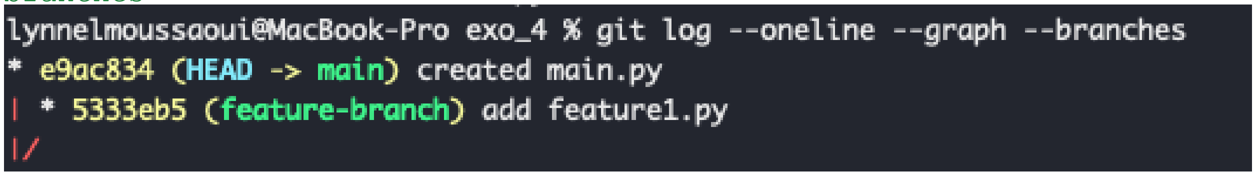
ii Mixed: changes are put in working dir

iii Hard:discard changes as a whole

4 : Git rebase

Task

1. Create a new branch named `feature-branch` and switch to.
    a. Can you do it in one git command ? using git checkout
2. In the `feature-branch`, create a file called `feature1.py`, write some content into it, and commit the file.
3. Switch back to the `main` branch.
4. Create a new file on the `main` branch called `main.py`, add content to it, and commit the file.
5. Check the commit history before your next step: `git log --oneline --graph --branches`

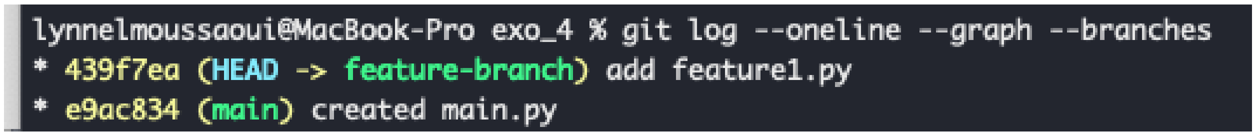


```
lynnelmoussaoui@MacBook-Pro exo_4 % git log --oneline --graph --branches
* e9ac834 (HEAD -> main) created main.py
| * 5333eb5 (feature-branch) add feature1.py
|/
```

6. **Rebase Step 1:** Switch back to the `feature-branch` and use the `git rebase main`
    a. What do you think rebase is doing? Which branch will be the base ?

       Rebase is taking commits from feature-branc and putting them on top of main, the base branch is main

    b. Check the status and commit history using: `git log --oneline --graph --branches`



```
lynnelmoussaoui@MacBook-Pro exo_4 % git log --oneline --graph --branches
* 439f7ea (HEAD -> feature-branch) add feature1.py
* e9ac834 (main) created main.py
```

c. What has changed? We had diverging branches, but after the rebase they became linear

d. What's the difference between this and merging `main` into `feature-branch` ? Merge shows the branches getting merched, there exists a merge commit. Rebase re writes the history to make it look linear as if no branch was there, the hashes of the items is also changed

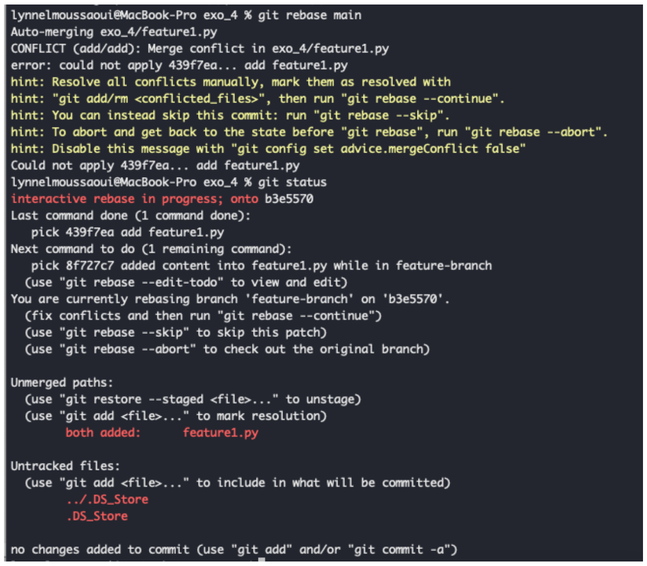
7. **Rebase with Conflicts**:

   a. On the `main` branch, modify the content of `feature1.py` and commit the change.

   b. Switch back to the `feature-branch` and modify `feature1.py` as well, making a conflicting change, then commit it.

   c. Attempt to rebase the `feature-branch` onto `main` again using `git rebase main`. This should result in a conflict.

   d. Use `git status` to see which files are in conflict.



```
lynnelmoussaoui@MacBook-Pro exo_4 % git rebase main
Auto-merging exo_4/feature1.py
CONFLICT (add/add): Merge conflict in exo_4/feature1.py
error: could not apply 439f7ea... add feature1.py
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
hint: Disable this message with "git config set advice.mergeConflict false"
Could not apply 439f7ea... add feature1.py
lynnelmoussaoui@MacBook-Pro exo_4 % git status
interactive rebase in progress; onto b3e5570
Last command done (1 command done):
   pick 439f7ea add feature1.py
Next command to do (1 remaining command):
   pick 8f727c7 added content into feature1.py while in feature-branch
  (use "git rebase --edit-todo" to view and edit)
You are currently rebasing branch 'feature-branch' on 'b3e5570'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
        both added:      feature1.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ../.DS_Store
        .DS_Store

no changes added to commit (use "git add" and/or "git commit -a")
```

8. **Resolving Rebase Conflicts**:

   a. Open `feature1.py` and manually resolve the conflict by editing the file.

   b. After resolving the conflict, use `git add` to mark the conflict as resolved.

   c. Complete the rebase by running `git rebase --continue`.

   d. If you want to stop the rebase process and undo the changes, use `git rebase --abort`.

   e. Try using this command before completing the rebase.

      i. What is the current state of your repo ? in a detached HEAD state with unresolved conflicts waiting for manual resolution.

Exercise

Main

ii. Can you repeat the rebase and finish it correctly now ? I rsolved the conflicts staged he file and re rean git rebase to complete the rebase

9. After completing the rebase, check the commit history with `git log --oneline --graph --branches`
    a. What does the commit graph look like after the rebase? linear
    b. How did the changes from `main` and `feature-branch` have been combined ? into a linear history
    c. How does it compare to the graph when you merge branches? In rebase the graph looks like one continuous line. In merge it shows two diverging lines joined by a merging commit
10. Create a few more commits on both `main` and `feature-branch`.
11. Experiment with using `git rebase --interactive` (or `git rebase -i`) to reorder, squash, or modify the commits during the rebase.
    a. Try squashing a commit.
    b. What does squashing do to the commit history? Combined commits, I combined 2 commits
    c. Modify a commit during an interactive rebase.
    d. What steps do you need to take to amend an old commit during rebase? Marked the commit with edit while in the interactive base. Git stopped at that commit, did commit –amend, changed the commit message, saved then quit, and finally I did git rebase –continue
12. Use `git log --oneline --graph --branches` to inspect the commit history and see how the rebase has altered it.

Stretch Task

1. **Interactive Rebase**: Use `git rebase -i HEAD~3` to rebase the last three commits interactively. Experiment with the following options:
    a. Reword a commit message. Changed pick to reword
    b. Squash two commits together. Changed pick to Squash
    c. Drop a commit. Changed pick to Drop
    d. What effect does each of these actions have on the commit history? The one I dropped is gone from history, and the one I squashed are under one commit with the new message I chose

2. **Rebase vs Merge**:
    a. Create another new branch, make several commits.
    b. Go back to main and create a few commits as well.
    c. Merge `main` into your new branch using `git merge`.
    d. Next, re-create the branch, and instead of merging, use `git rebase main` and then `git merge`. Compare the commit history in both cases. How does the

history differ between merge and rebase? Rebase the logs are linear, merge preserves every step

3. **Skipping a Commit During Rebase**: Create a situation where there will be a conflict in your rebase like step 7.
    a. Perform a rebase.
    b. During the rebase, git should break on the conflicting commit.
    c. Use the `git rebase --skip` command to skip this commit that you don't want to apply.
    d. How does this affect the commit history? The commit I skipped is dropped, any changes from it are not applied