Git Laboratory Exercise 1

Anthony Ayli 220993

1 Exercise 1

1.1 Question 3-a

The `.git` directory is used to store all of Git's metadata, version history, configuration settings, and tracking information for the repository.

1.2 Question 6-a

Staging area.

1.3 Question 11-b

The `git diff` command now gives no output because it only shows differences between the working directory and the staging area — since I added the changes to staging, there are no differences between these two areas. To see a diff of things in the staging area (staged changes), I need to use `git diff --staged` or `git diff --cached`.

1.4 Question 14-a

At least 4 characters.

1.5 Exercise 1 Stretch Task

1.5.1 Question 3

`git rm` → deletes the file and stages the deletion. On the other hand `rm` / `del` → deletes the file, but the deletion is only shown as unstaged, and I must stage it manually before committing.

1.5.2 Question 5a

It does not automatically recognize it as a rename. It just sees: one file deleted, one new file added (but not staged yet). Without staging both, the commit will not record the rename correctly — I will end up with either just a deletion.

1.5.3 Question 5-c

No, Git did not work it out during status/staging. But yes, Git can work out the rename when showing diff.

1.5.4 Question 7-a

- `git show --stat`: Shows the details of the most recent commit plus a per-file summary of changes.
- `git log --stat`: Each commit in the log will include the file-level change summary.
- `git diff --stat`: Instead of showing all line-by-line differences, it summarizes what files were changed, how many lines were added/removed, and totals.

1.5.5 Question 8

What it shows: A patch-style diff of everything that changed between the two commits. By default, line-by-line changes for every file touched in that range.

2 Exercise 2 Stretch Task

2.1 Question 4-a

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

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

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

2.2 Question 6-a

No I got:

```
U       debugging.py
U       testing.py
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
```

Note: I edited two files on both the master branch and on my_first_branch. When I merged and got a conflict, I resolved it by keeping the code from my_first_branch in the first file, and the code from master in the second file.

2.3 Question 9-b

When I tried to merge, Git gave me a modify/delete conflict because one branch had changes to the file while the other branch had deleted it. To resolve it, I had to decide whether I wanted to keep the file with the changes or remove it completely. Once I made my choice, I staged the result and committed the merge.

Important Note: While creating the situation in question 9 I accidentally removed the readme file so I reverted the commit after I made my choice.

3 Exercise 3

3.1 Question 6-b

The changes I made in step 5 were completely discarded. When I used `git checkout -- file3.py`, Git restored the file back to its last committed version, so the new line I added was lost and is no longer in the working directory.

3.2 Question 6-c

Yes, I could have achieved the same result with a different command. The message from `git status` suggests using: `git restore file3.py`

3.3 Question 8-a

When I used the `git revert` command, Git created a new commit that undid the changes from my previous commit. In contrast, when I use `git checkout` on a file, it just discards my local changes in the working directory without creating a new commit.

3.4 Question 8-c

When I looked at the history, I noticed that `git revert` created a new commit instead of removing the old one. This new commit has its own commit ID and shows the inverse of the changes I made in the original commit, effectively undoing it while still keeping a full record of what happened.

3.5 Question 10-b

When I ran `git reset --soft HEAD^`, Git moved the branch pointer back one commit, but it left all the changes from that commit staged in the index. That means my commit was undone, but the edits in file3.py and file4.py are still staged and ready to be committed again.

3.6 Question 10-c

No, I don't see my previous commit anymore in the log because the branch pointer no longer points to it. However, the changes from that commit haven't been lost — they are still in the staging area.

3.7 Question 10-d

To avoid losing the commit after doing a reset, I could have created a backup branch before resetting, or I could have used `git revert` instead. `git revert` would have undone the commit while keeping it visible in the history.

3.8 Question 11-b

I used `HEAD` instead of `HEAD^` because I wasn't trying to move the branch pointer back to an earlier commit. Instead, I just wanted to reset the staging area relative to the current commit. In other words, `HEAD` means "compare against the commit I'm currently on" and only affect what's staged, not the commit history.

3.9 Question 11-c

Since I didn't specify a scope, Git used its default mode, which is `--mixed`. That scope means the changes stay in my working directory but are unstaged. So the edits are still there in the files, but they're no longer marked to be included in the next commit.

3.10 Question 12-b

Since I didn't give a commit, Git used `HEAD` as the default — it reset everything to the current commit.

3.11 Question 12-c

If I had done this right after step 9, my commit (with changes in file3.py and file4.py) would have been completely lost, and the working directory would have been wiped back to the previous commit.

3.12 Exercise 3 Stretch Task

3.12.1 Question 2-a

After I used `git checkout -- file5.py`, the file was restored to the last committed version. The changes I had made to it were discarded.

3.12.2 Question 2-b

In the working directory, Git replaced my edited copy of file5.py with the version from the latest commit, so the working directory now matches the repository's state for that file.

3.12.3 Question 2-c

The `HEAD` pointer does not move at all. It still points to the same latest commit — only the contents of the file in my working directory were reset to match that commit.

3.12.4 Question 4-a

The commit history grows — Git doesn't delete anything. Instead, it adds new commits on top that undo the effects of the last two commits. The original commits are still there in the history.

3.12.5 Question 4-b

The changes from those two commits are reversed. Git applies the opposite changes.

3.12.6 Question 4-c

The syntax `HEAD~2..HEAD` means "the range of commits starting from two commits before." So in this case, it selects the last two commits.

3.12.7 Question 4-d

Yes, we can use a similar syntax with `git reset`. For example, `git reset --hard HEAD~2` moves `HEAD` back two commits. But the key difference is:

- With revert, commits stay in history and are undone safely.
- With reset, commits can be removed from history.

3.12.8 Question 7-a

When I ran `git reset HEAD~3`, Git moved the branch pointer back three commits in history. The last three commits disappeared from the current branch's history, but their changes were left in my working directory and also unstaged. The commits were undone, but the actual file changes from those commits are still present in my working copy so I can review, restage, or recommit them if I want.

3.12.9 Question 7-b

To move my current branch back to anchor, I used `git reset --hard anchor`. This moved the branch pointer to the anchor commit and also updated both my staging area and working directory so that everything exactly matched the saved state at anchor.

4 Exercise 4

4.1 Question 1-a

Yes, I can do it in one command using: `git checkout -b feature-branch`

4.2 Question 6-a

When I ran `git rebase main` from feature-branch, Git took the commits from feature-branch and replayed them on top of the tip of main. That means main became the new base, and my feature commits were reapplied as if they happened after main's latest commit.

4.3 Question 6-c

After the rebase, the commit history changed — it now looks like a straight line. The commits from feature-branch were reapplied on top of main, so they have new commit IDs and appear "after" the commits in main. The history is cleaner and linear, without a merge commit.

4.4 Question 6-d

The difference compared to merging is that with a merge, Git would create a merge commit to tie the two histories together, preserving the branching structure. With rebase, Git rewrites history so that it looks like my feature work was developed directly on top of main from the start, avoiding a merge commit and keeping the history linear.

4.5 Question 8e-i

When I ran `git rebase --abort`, Git stopped the rebase process and rolled my repository back to the exact state it was in before I started the rebase. That means all of my commits and changes are safe, and the conflict is gone as if the rebase never happened.

4.6 Question 8e-ii

Yes, I can repeat the rebase now. If I run `git rebase main` again, I'll hit the same conflict, but this time I can resolve it manually in feature1.py, stage the fix with `git add`, and then finish the process with `git rebase --continue`. That will successfully apply my feature branch commits on top of main.

4.7 Question 9-a

After I completed the rebase, my commit graph turned into a straight line. All the commits from my feature-branch now sit on top of the commits from main, and there are no merge commits.

4.8 Question 9-b

The changes from both branches were combined by replaying my feature-branch commits on top of main's commits. That way, it looks like I built my feature directly after the work that was already done in main.

4.9 Question 9-c

Compared to merging, this looks much cleaner. If I had merged instead, the history would show both branches diverging and then a merge commit tying them together.

4.10 Question 11-b

When I squashed a commit during the interactive rebase, Git combined it into the previous commit. In the commit history, this reduced the total number of commits — instead of seeing two separate commits, I saw one commit that contained the combined changes.

4.11 Question 11-d

To amend an old commit during a rebase, I marked that commit with `edit` in the rebase todo list. Then, when Git stopped at that commit, I made my changes, staged them with `git add`, ran `git commit --amend` to update the commit, and finally continued the rebase with `git rebase --continue`.

4.12 Exercise 4 Stretch Task

4.12.1 Question 1-d

Since I already tried rewording, squashing, and dropping in question 11, here's what I saw:

- When I reworded a commit message, the commit itself stayed the same in terms of changes, but its message was updated. The history still showed the same number of commits, just with a different description.
- When I squashed two commits together, Git combined them into a single commit. The history became shorter, with one commit that contained both sets of changes instead of two separate ones.
- When I dropped a commit, that commit disappeared completely from the history. Its changes were gone, as if the commit had never been made.

4.12.2 Question 2-d

When I compared the histories, I saw a clear difference between merge and rebase:

- With merge, my new-branch history showed a branch split and then a merge commit tying main and new-branch back together. This kept the full branching history.
- With rebase, my commits from new-branch were replayed on top of main. It looked like I had started working on branch after the latest commits from main.

4.12.3 Question 3-d

In the commit history, it looked like I had never made that skipped commit — its changes and message were gone, but the later commits remained intact.