**Git Lab**

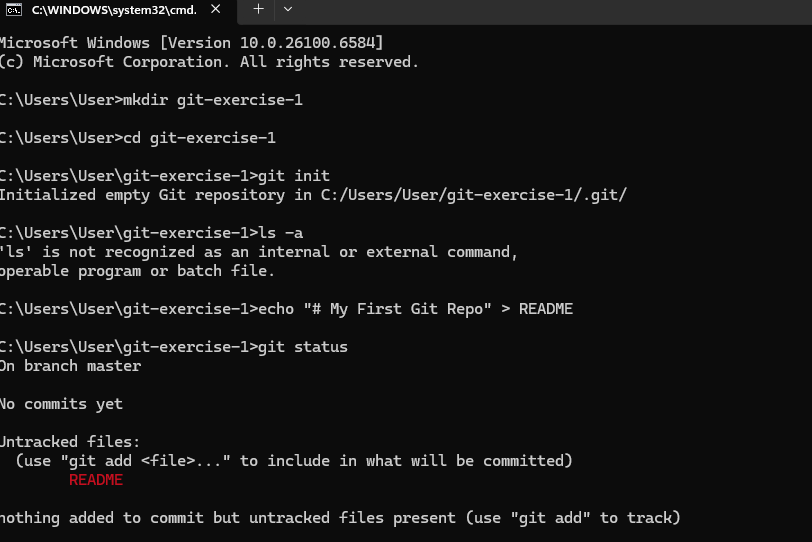
**Elie Haouchan**

**Github Repository:** [**https://github.com/eliexh/Git\_Lab.git**](https://github.com/eliexh/Git_Lab.git)

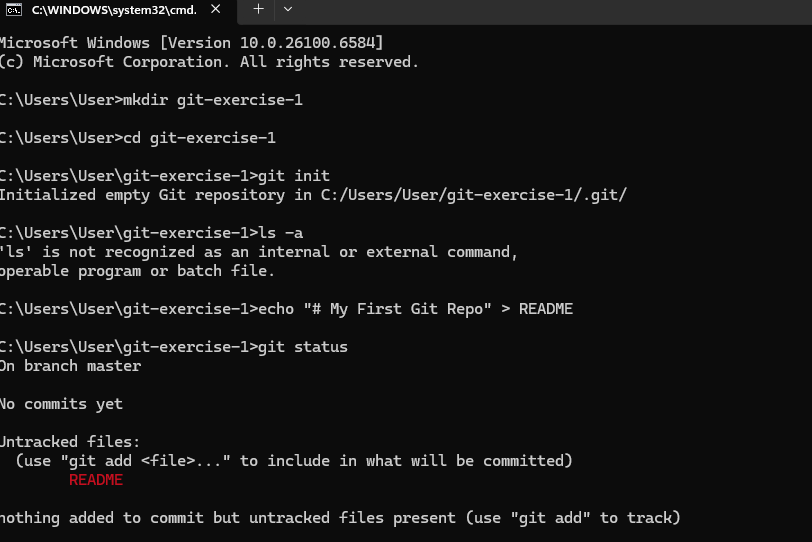
**Exercise 1 - Tracking Files**

Main Task

1. Create a new directory and change into it.



2. Use the init command to create a Git repository in that directory.



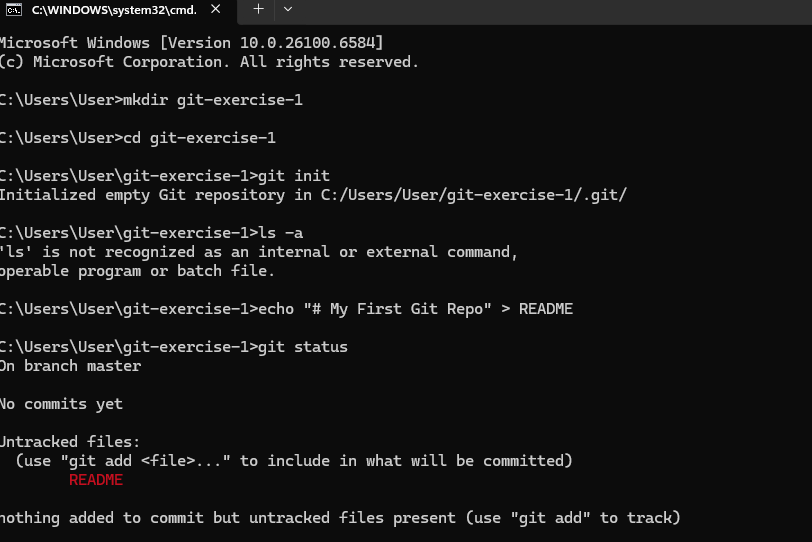
3. Observe that there is now a .git directory.

a. What is it used for ?

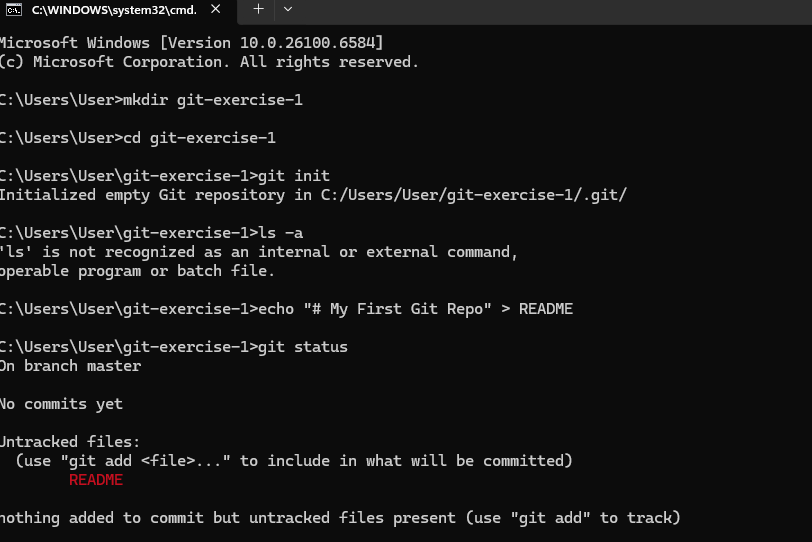
The .git directory contains all the internal data Git uses to manage the repository, such as:

* The entire commit history
* Configuration settings
* References (branches, tags)
* The staging area (index)
* Objects like commits, trees, blobs

4. Create a README file.



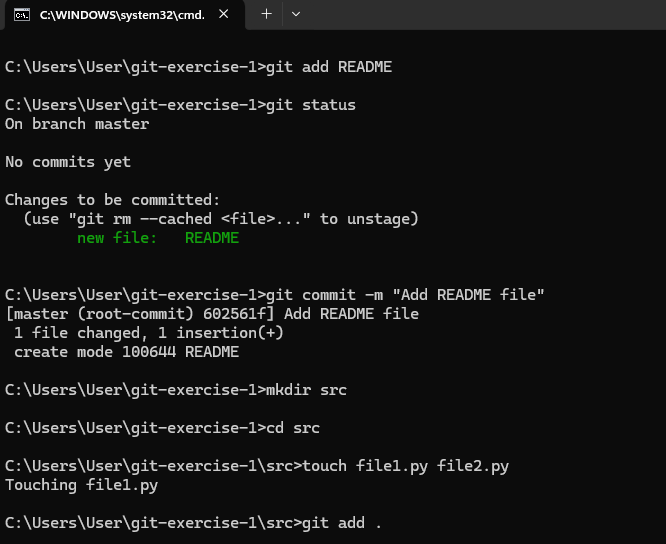
5. Look at the output of the status command; the README you created should appear as an untracked file.



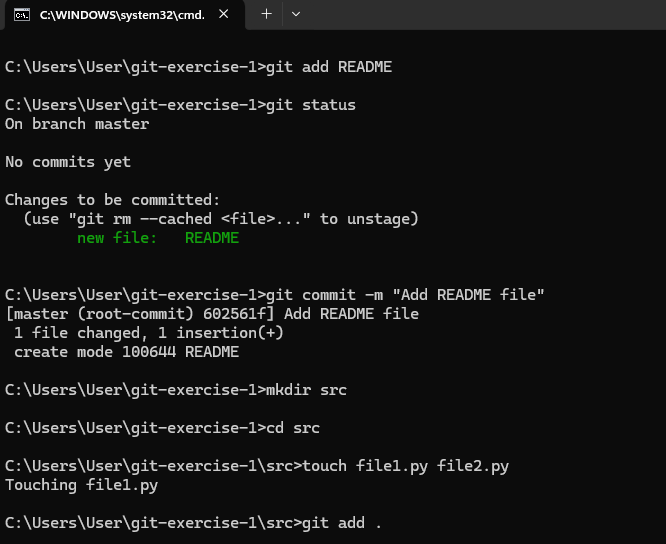
6. Use the add command to add the new file to the staging area. Again, look at the output of the status command.

a. In which stage does the file appear ?

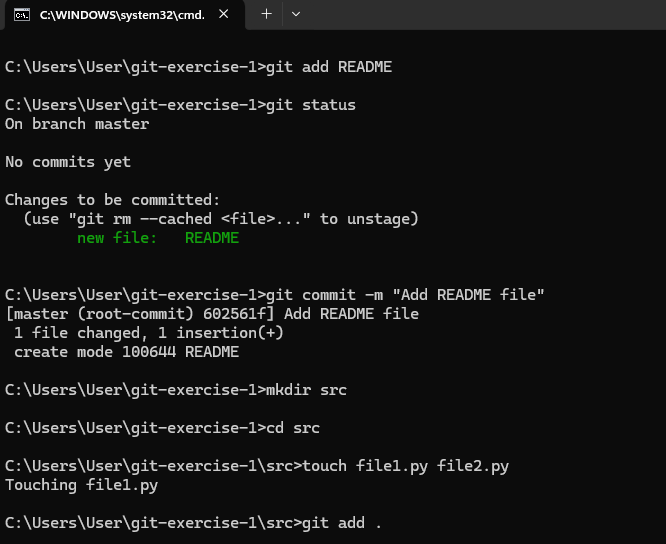
The file appears in the staging area (also known as the index). It’s now ready to be committed. Git is tracking it and waiting for us to confirm with a commit.



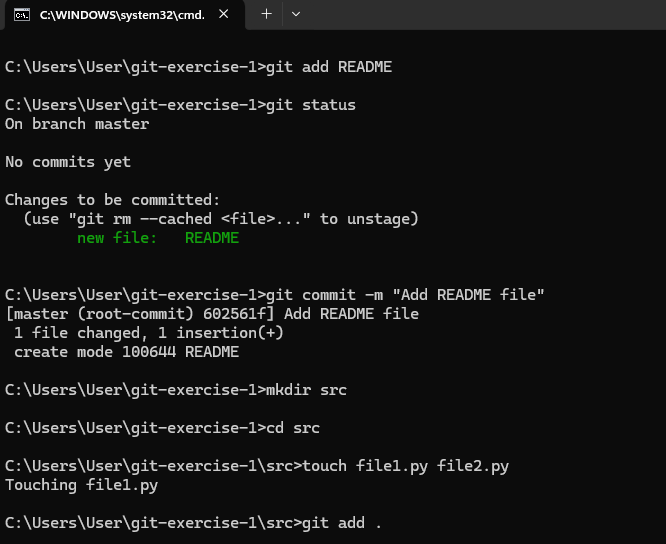
7. Now use the commit command to commit the contents of the staging area.

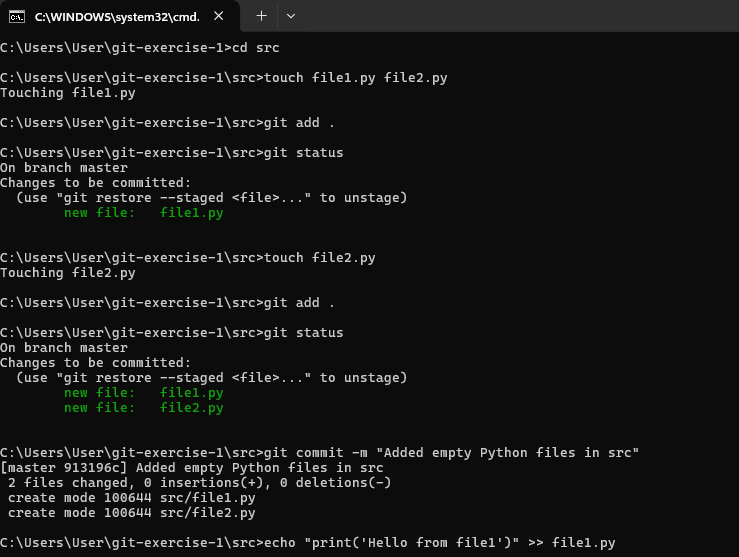


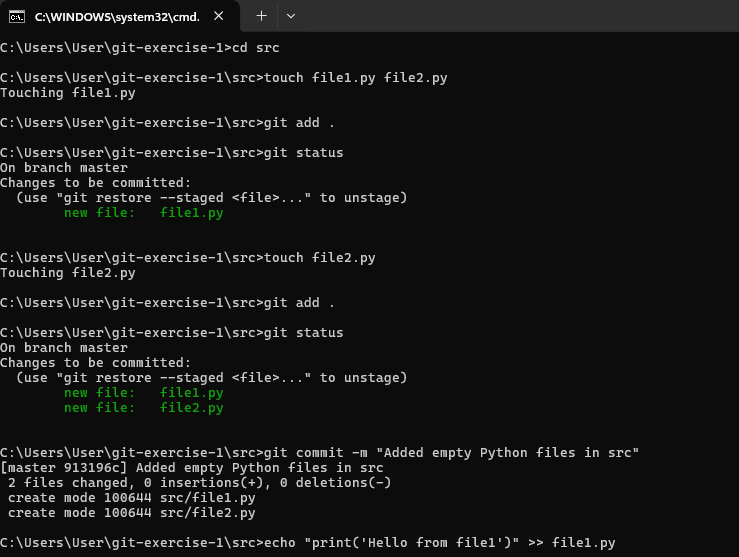
8. Create a src directory and add to it two new empty files: file1.py and file2.py.



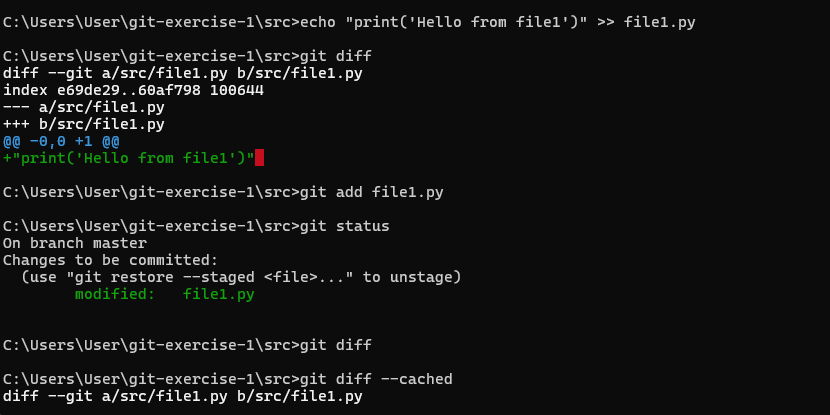
9. Use the add command on the directory, not the individual files. Use the status command. See how both files have been staged, then Commit them.



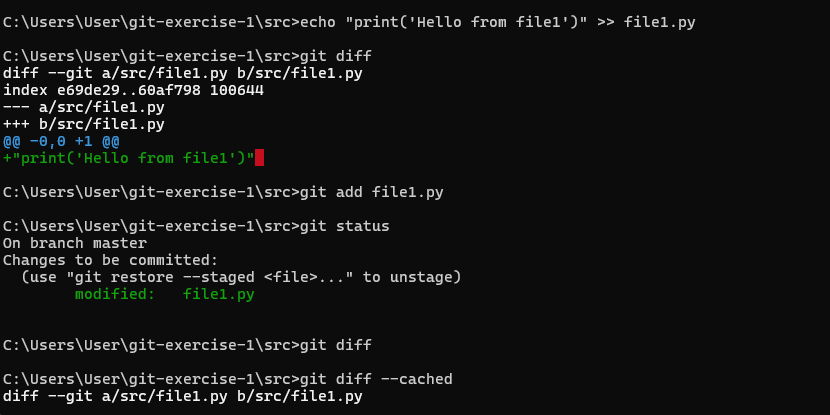




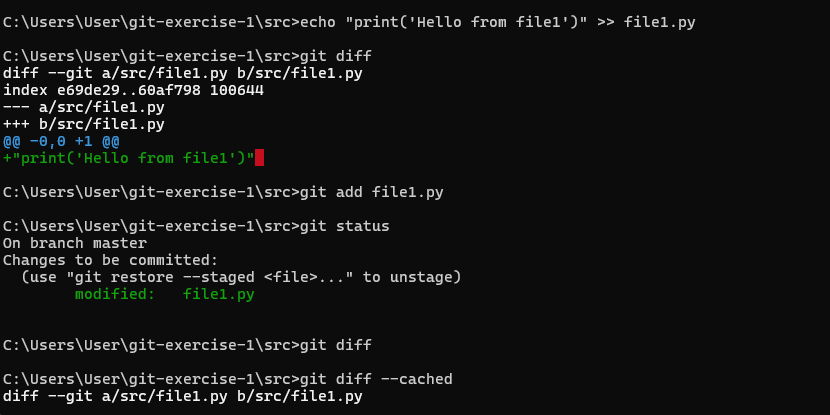
10. Make a change to file1.py. Use the diff command to view the details of the change.



11. Next, add the changed file, and notice how it moves to the staging area in the status output.



1. Observe that the diff command you did before using add now gives no output.

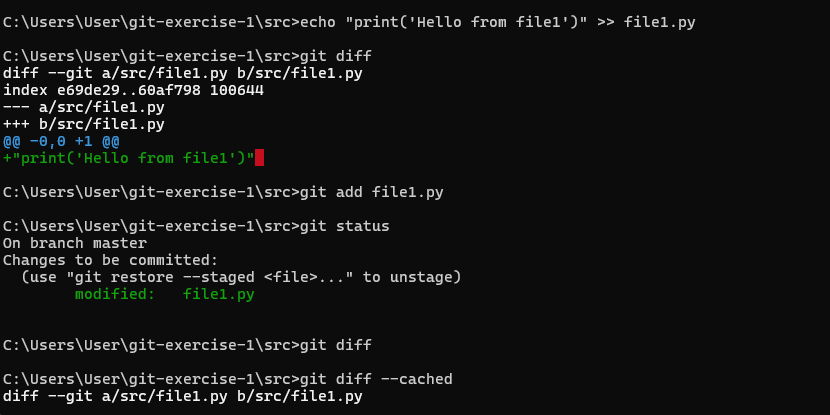


1. Why not? What do you have to do to see a diff of the things in the staging area?

Because git diff by default shows **unstaged** changes only — and you've already staged the changes.

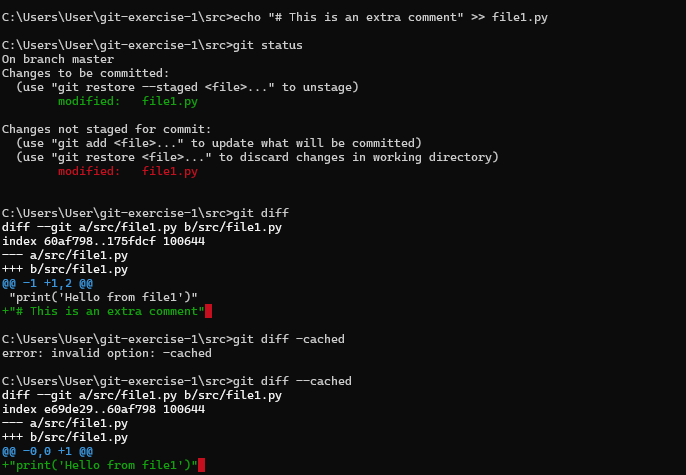
To see what’s in the staging area (compared to the last commit):

git diff --cached

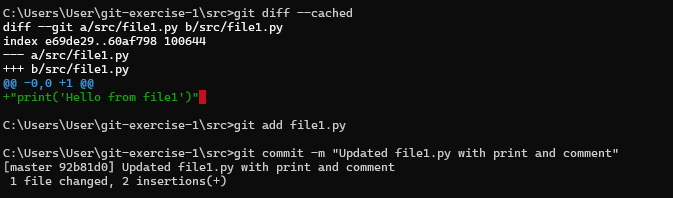


12. Without committing, make another change to the same file you changed in step 10.

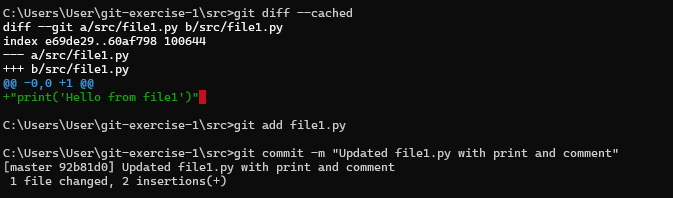
Look at the status output, and the diff output.



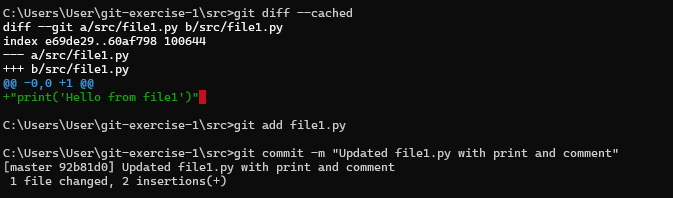
a. Notice how you can have both staged and unstaged changes, even when you’re talking about a single file.



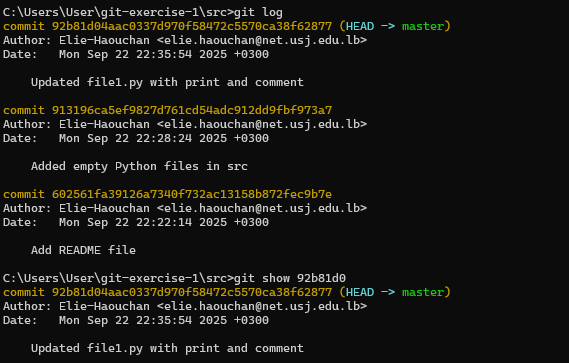
b. Observe the difference when you use the add command to stage the latest round of changes.



1. 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.

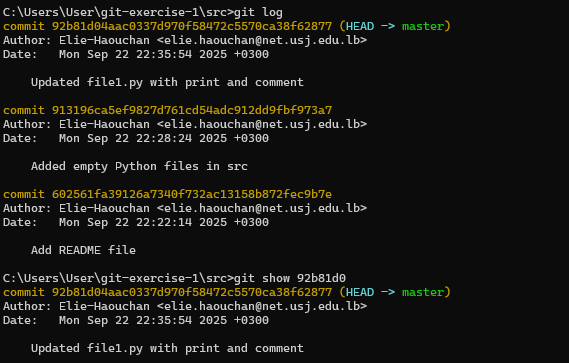


14. Use the show command to look at an individual commit.

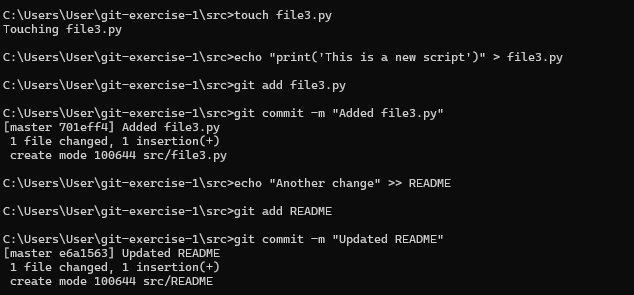
a. How many characters of the commit identifier can you get away with typing at a minimum?

Usually 7 characters is enough (e.g., 3f1d2b3), but:

* Git allows short hashes as long as they are unique
* If two commit hashes share a prefix, you’ll need more characters



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



**Stretch Task**

3. Look at the status. Compare it to the status output you had after using the Git built-in rm command. Is anything different?

**Key Difference:** Manual deletion requires git add to stage the removal; git rm does it for you.

5.a. Will you get the right outcome if you commit at this point?

No — Git sees:

* One file deleted
* One new untracked file

If we commit now, Git would:

* Record the deletion of README.md
* Ignore readme.txt

5.b. How to fix this?

We need to stage both the removal and the new file so Git can associate the rename.

5.c. Did Git work out the rename?

Yes, but only after we staged both changes.

7.What does --stat do?

It gives a summary of file changes per commit.

8. Use git diff between two commits

This shows the changes made between those two commits:

* All additions, deletions, and edits
* Just like git diff for working changes, but now for historical commits

**Exercise 2 – Git Branches**

8.a. What is a "fast-forward" merge?

A fast-forward merge happens when:

* The branch you're merging has all the commits ahead of your current branch
* Git simply moves the master branch pointer forward

No merge commit is needed.

* 1. Check git log again: There’s **no merge commit**, because it was a fast-forward merge.

c. View commit graph again:

git log --graph --oneline

Still a linear history — no branching arrows, because Git fast-forwarded master.

**Exercise 3 – Undoing Changes in Git**

6.b. What happened to the changes made in step 5?

They were lost.

Because they were not staged or committed, git checkout -- file3.py simply restored the last committed version, discarding our local edits.

* 1. Could you have used a different command?

Yes — as per the hint in git status, we can also use:

git restore file3.py

This is the modern equivalent of git checkout -- file3.py and is preferred for clarity.

8.a. What does revert do compared to checkout?

| Command | Effect |
| --- | --- |
| checkout or restore | Discards uncommitted changes (working directory) |
| revert | Creates a new commit that undoes the effect of a previous commit — keeping history intact |

c. What do you notice about the new commit?

* It’s a new commit
* Message says "Revert "Add file4.py" — auto-generated by Git
* It undoes the changes, but keeps the full history visible

10. Use git reset --soft HEAD^

b. What happened?

* The most recent commit was undone, but:
* All the changes from that commit are still in the staging area
* HEAD now points to the previous commit

c. Do you still see your previous commit?

git log --oneline

No — the commit is no longer in the log because HEAD was moved back and the commit is now dangling.

(We could recover it using git reflog, but it’s not visible in log anymore.)

d. ❓ How could you have avoided losing the commit?

Before doing git reset, we could have:

* Copied the commit hash (git log)
* Tagged it: git tag backup-before-reset
* Or used git reflog after reset to recover the dangling commit

11. Use git reset HEAD

b. Why HEAD instead of HEAD^?

Because we already moved HEAD back with --soft HEAD^, we’re now working with the current HEAD — no need to go back again.

c. What scope did Git use by default?

Git used the --mixed scope by default.

So git reset HEAD is equivalent to:

git reset --mixed HEAD

12: Use git reset --hard

a. What happens?

* Working directory is reset to match the current commit
* All uncommitted changes are discarded
* Staging area is cleared

✅We’re now fully clean — like nothing ever happened.

b. What did Git use as the default value?

If you don’t give a commit ref, Git defaults to:

git reset --hard HEAD

So it resets to the current HEAD.

c. What if you had run --hard right after Step 9?

We would have:

* Lost our last commit permanently (unless recovered via reflog)
* Lost all changes in file3.py and file4.py
* No way to recover unless backed up or committed elsewhere

git reset --hard is destructive — use with caution.

**Stretch Task**

2. a. What happens to the file after using checkout?

The file is restored to its last committed state.  
Any unsaved edits (not staged or committed) are discarded.

In this case, the line we added to file5.py is gone.

b. What does it do to the working directory?

It modifies the working directory to match the file as it exists in HEAD (i.e. the last commit).

* Any uncommitted changes are erased
* The file becomes clean (i.e. unchanged in git status)

c. What happens to the HEAD pointer?

Nothing.

git checkout -- <file> (or git restore) is a working directory command. It does not move HEAD and doesn't touch the commit history or branch pointer.

4. Use git revert HEAD~2..HEAD

### a. What happens to the commit history?

Git creates **three new revert commits**, one for each commit being reverted.

### b. How are the changes handled?

Each revert creates a new commit that **applies the inverse** of the original commit.

This is **non-destructive**, safe, and used in shared repositories.

### c. What does HEAD~2..HEAD mean?

In Git range syntax:

* HEAD~2 means "2 commits before HEAD"
* HEAD~2..HEAD means: **from the commit after** HEAD~2 **up to and including** HEAD
* So this range includes: HEAD~1, HEAD

But Git's revert includes the **starting commit**, so this will revert:  
→ HEAD~2, HEAD~1, and HEAD

### d. Can you use this same syntax with git reset?

Yes! Git's reset command **also supports ranges** like:

git reset --hard HEAD~3

But be aware:

* git reset **moves the HEAD and branch pointer**
* It’s **destructive** depending on the scope (--soft, --mixed, --hard)

7. Use git reset HEAD~3

### a. What happened?

* Your branch pointer moved back 3 commits
* You lost the 3 revert commits from the history (but changes are still in your working dir)
* Git **unstaged** the changes

### b. How to move your branch back to anchor?

To move back to the **exact commit** that anchor points to, run:

git reset --hard anchor

This will:

* Move our current branch to the anchor commit
* **Wipe any uncommitted changes**
* Fully restore our staging area and working directory

**Exercise 4:**

1. Create a new branch named feature-branch and switch to it

git checkout -b feature-branch

This does two things:

Creates a new branch called feature-branch from our current branch (likely master)

Switches to it immediately

6. Rebase Step 1

### a. What is rebase doing?

Rebase is:

* Taking the commits **on your current branch (**feature-branch**)**
* And **replaying them** on top of the main branch

In other words, Git:

1. Temporarily removes our commit(s)
2. Moves feature-branch to the tip of main
3. Re-applies our commit(s) one by one

This creates a **linear history** as if our feature work started after main.py was added.

### c. What has changed?

* feature-branch’s commit was **rebased** (replayed) on top of main
* The history is now **linear**
* The commit hash of feature1.py has changed because Git created a new commit with the same content but a new parent (main's latest commit)

d. What’s the difference between merge and rebase?

| Merge | Rebase |
| --- | --- |
| Keeps full history (with branches and merge commits) | Rewrites history to be linear |
| Easier for collaboration (preserves context) | Cleaner history (looks like a straight line) |
| Can result in merge commits | No merge commits |
| Does not change commit hashes | Rewrites commit hashes |
| Example: git merge main | Example: git rebase main |

### 8.e. Try git rebase --abort before finishing

i. What is the current state of your repo?

After aborting:

* We’ll be back on feature-branch
* Your commits are intact
* The rebase has **not** been applied

ii. Can you repeat the rebase and finish it correctly now?

Yes! Just re-run:

git rebase main

And follow the same conflict resolution steps:

* Edit the file
* git add
* git rebase --continue

9. After completing the rebase, check the commit history:

git log --oneline --graph --branches

### a. What does the commit graph look like after the rebase?

* The graph will show a **linear history**.
* Our feature-branch commits will appear **on top of** the latest commits from main.
* Unlike merges, there will be **no merge commit** — just a clean sequence of commits.

### b. How did the changes from main and feature-branch have been combined?

* Git effectively **re-applied** your feature branch commits as if they were based on the current main head.
* This means our feature commits come **after** the latest main commits.
* This avoids the "merge commits" and keeps history linear.

### c. How does it compare to the graph when you merge branches?

* **Merging** creates a commit graph with a **merge commit** showing the branch point.
* The history is a **Directed Acyclic Graph (DAG)** with branches and joins.
* **Rebasing rewrites history** to avoid this, making it appear as if all commits happened sequentially.

11. Experiment with interactive rebase

Start an interactive rebase on your current branch, rewriting the last few commits (e.g., last 3 commits):

git rebase -i HEAD~3

a. Try squashing a commit

* In the editor that opens, we changed the word pick to squash on the commit(s) we want to combine into the one above.
* Save and close the editor.
* Git will then combine those commits into a single commit and prompt us to edit the commit message.

b. What does squashing do to the commit history?

* It combines multiple commits into one.
* The commit history becomes cleaner and more concise.
* Useful for grouping related changes.

c. Modify a commit during an interactive rebase

* Change pick to edit (or e) on the commit you want to modify.
* When Git pauses at that commit, you can amend it:

# Make your changes to files

git add <files>

git commit --amend

* Then continue rebase:

git rebase --continue

d. What steps do you need to take to amend an old commit during rebase?

1. Use git rebase -i and mark the commit as edit.
2. When stopped, make the changes.
3. Stage them (git add).
4. Amend the commit (git commit --amend).
5. Continue the rebase (git rebase --continue).

12. Inspect commit history after rebase

git log --oneline --graph --branches

* You’ll see the commits reordered or combined based on your interactive rebase edits.
* This helps maintain a clean, understandable history before merging to main.

**Stretch Task**

1. Interactive Rebase: git rebase -i HEAD~3

Effects on Commit History

| Action | Effect |
| --- | --- |
| Reword | Changes only the commit message, not the contents. |
| Squash | Combines two (or more) commits into one. |
| Drop | Removes the commit entirely (it’s like it never existed). |

1. Rebase vs Merge

d.With rebase:

The commit history is linear.

No merge commit is needed when rebasing.

Useful for clean history, but be cautious — rebasing rewrites commit IDs.

3. Skipping a Commit During Rebase

d. Effects on History

The **skipped commit is lost** in this branch.

If it introduced a feature or fix, it's now gone.

Use it **only** when the commit can be discarded safely.