**Hadi cheayto**

**Matriule: 221008**

**Master intelligence artificielle**

**Course: MLOPS**

**GIT LAB:**

**Exercice1:**

**3a:**   
The .git directory is a hidden folder created by Git, a distributed version control system, within the root of a project. It serves as the local repository for that project, containing all the necessary information for Git to track changes, manage versions, and facilitate collaboration.

**6a: In which stage does the file appear?**  
After git add, it appears in the staging area as “changes to be committed.”

**11a/b:**

* After i stage, git diff shows nothing because it compares working tree vs. index, and i have no unstaged changes now.
* To see what’s in the staging area vs. the last commit, we can use: “git diff –staged”

**14a: Minimum characters of the commit id?**  
As many as needed to be unique in this repo, with a minimum of 4 hex chars. In small repos, 7 is typical; if ambiguous, type more.

**Stretch Task:**

**3- Difference:** “git rm” stages removal immediately; OS delete requires “git add -A (or git rm afterward)” to stage.

**5-a/b/c**

* No, committing at this point won’t give the right outcome , since the file is untracked we should stage the file before commiting
* to Make status show it won’t lose the file we should Add them using git add -A.
* If contents are similar enough, Git will show it as a rename.

**7-a -** It prints a per-file summary of changes (added/removed lines) and a total at the end.

**Exercice3:**

**6b.** They were **discarded** from the working directory (reverted to last committed version).

**6c.** Yes: “git restore ex3\file3.py” does the same

**8a.**

* revert **adds** a new commit that undoes the changes → history preserved.
* “checkout -- <file>” just resets the file in your working tree (no new commit).

**C -**I saw a **new** “Revert” commit that removes file4.py.

**10b.** --soft moved HEAD back one commit but kept all changes staged. my commit content is now ready to recommit.

**10c.**   
No the branch tip moved back. That commit is no longer referenced (but still recoverable via reflog).

**10d.** we can create a branch before the reset

**11a.**   
Now the changes moved from staged → unstaged. Files are modified in working tree but not staged.

**11b.**   
We already rewound to the parent; using HEAD resets index to match current HEAD, simply un-staging files.

**11c.**   
“git reset” default is mixed (reset index only).

**12a.**   
Both working directory and index are reset to HEAD. Local changes are discarded. Commit history stays at current HEAD.

**12b.**   
If no commit is specified, Git uses HEAD.

**12c.**

I will lose the work in file3.py and file4.py immediately (both unstaged and staged) hard reset would erase them from the working copy.

Stretch Task:

**2-**

1. Reverted to last committed content.
2. Becomes clean for that file (no local mods).
3. Unchanged “checkout -- <file>” doesn’t move HEAD.

**4-**

1. New **revert commits** are added (no history rewrite).

**b.** Each reverted commit is **undone** in new commits.  
**c.** Range from “two before HEAD **exclusive**” up to HEAD **inclusive** (i.e., last two commits).  
**d.** Yes, but reset uses a **single** target commit; we can’t pass a range though i can point to HEAD~2. (To “drop” a range I have to reset to the commit **before** the range.)

**7-**

**a.** Branch tip moved to an older commit; commits ahead are dropped from the branch.  
**b.** Yes , “git reset --hard anchor“.  
**c.**

* Soft: “git reset --soft anchor” (keep everything staged)
* Mixed: “git reset anchor” (unstage, keep changes)
* Hard: “git reset --hard anchor” (discard local changes)

**Exercice 4:**

**1a.** Yes “git checkout -b new-branch” (create + switch).

**6a.** It **replays** your feature-branch commits **on top of** main. New base = main.

**6c.** the feature commit now sits **after** the latest main commits (linear history).

**6d. Difference vs merge?**

* **Merge**: keeps both histories and makes a merge commit (non-linear).
* **Rebase**: rewrites your branch so it looks like it was created after main’s latest commits (linear)

8.e i  
After “git rebase –abort”, the feature-branch is back to the exact commits and content it had before i started the rebase , the conflict is gone and the branch history is unchanged.

8.eii. Yes. i can re-run: “git rebase main”

**9a.** Linear (feature commits on top of main).

**9b.** the feature commits were **rebased on top of** main.

**9c.** Merge shows a branch/fork + merge commit; rebase shows a straight line.

**11b.** Combines multiple commits into one (cleaner history).

Stretch Task:

**1-d)**

* **Reword:** message changed; hash changed; number of commits unchanged.
* **Squash:** reduces commit count; combines diffs & messages; hashes after that point change.
* **Drop:** removes changes introduced by that commit from the branch history; reduces commit count; hashes after change.

**2-d)**

**Normal Merge:**  
The history shows two lines coming together. Git makes a merge commit to show exactly when the two branches were joined. This is useful when i want to keep a full record of how the work was done and when it was merged.

**Fast-Forward Merge:**  
If the main branch didn’t change, Git just moves the main branch forward to where the feature branch is. No extra commit is made, and the history stays in a straight line. It looks like all the commits was made directly on main.

**Rebase:**  
Rebase takes the feature branch commits and "replays" them on top of the latest main branch. The history becomes one straight line, but Git gives my commits new IDs because it rewrites them. It makes the log look very clean but changes the history.

**3-d)**

When you use “git rebase -skip”, Git just leaves out the commit that caused the conflict.  
I notice that commit B disppeared form my history.