**Name: Fatima Fadel**

**Id: 233960**

**Subject: MLOPS**

**Exercise 1 - Tracking Files**

1. Create a new directory and change into it.

Ans: Mkdir git\_assignment

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

Ans: git init

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

a. What is it used for ?

Ans: The .git directory contains all the metadata and history of your project, including commits, branches, tags, and staging information. It is the database that makes a project a Git repository.

4. Create a README file.

Ans: echo "# My First Git Project" > README

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

an untracked file.

Ans: git status

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

of the status command.

Ans: git add README

a. In which stage does the file appear ?

Ans: It now appears under “Changes to be committed.”

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

Ans: git commit -m "Add README file"

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

Ans: mkdir src

Set-Content -Path src\file1.py -Value 'print("Hello from file1")'

Set-Content -Path src\file2.py -Value 'print("Hello from file2")'

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.

Ans: git add src

git status

git commit -m "Add src directory"

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

Ans: Add-Content -Path src\file1.py -Value 'print("This is a new line in file1.py")'

git diff

output: ---> diff --git a/src/file1.py b/src/file1.py

index f5c6bf5..82e0890 100644

--- a/src/file1.py

+++ b/src/file1.py

@@ -1 +1,2 @@

 print("Hello from file1")

+print("This is a new line in file1.py") <---

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

output.

Ans: git add src/file1.py

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

Ans: no output.

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

Ans: git diff shows differences between the working directory and the staging area, and there are no un-staged changes left.

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

at the status output, and the diff output.

Ans: Add-Content -Path src\file1.py -Value 'print("Another change before committing")'

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

talking about a single file.

Ans: git status

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

of changes.

Ans: git add src/file1.py

git diff → empty (nothing unstaged)

git diff --staged → shows all changes that are now staged and ready to commit.

c. Finally, commit them. You should now have started to get a feel for the staging

area.

Ans: git commit -m "Update file1.py with additional changes"

13. Use the log command in order to see all of the commits you made so far.

Ans: git log

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

Ans: commit dfef9a4f2548ea31cb1994cc42716ef4905fbadf

show

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

minimum?

Ans: 4 characters

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

Ans: Set-Content -Path newfile.py -Value 'print("This is a new file")'

git add newfile.py

git commit -m "Add newfile.py"

**Stretch Task**

1. Use the Git rm command to remove a file. Look at the status afterwards. Now commit the deletion.

Ans: git rm src/file2.py

git status

git commit -m "Remove file2.py using git rm"

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.

Ans: del src\file1.py

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.

Ans: git status

output: 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:    src/file1.py

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.

Ans: git mv README README.md

git status

git commit -m "Rename README to README.md using git mv"

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?

Ans: ren README.md README.txt

git status

output: 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.md

Untracked files:

  (use "git add <file>..." to include in what will be committed)

        README.txt

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

Ans: Git thinks I deleted a file and created a new one, not a rename.

b. Work out how to get the status to show that it will not lose the file, and then commit.

Ans: git add -A

git status

git commit -m "Rename README.md to README.txt using OS command"

c. Did Git at any point work out that you had done a rename?

6. Use git help log to find out how to get Git to display just the most recent 3 commits.

Ans: git log -3

7. Try using --stat option with show command. Test it with log and diff commands.

a. What does it do ?

Ans: git show --stat

git log --stat -3

git diff --stat

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.

Ans: git diff fb517 ffd3a

**Exercise 2**

**Main Task - Git Branches**

1. Run the status command. Notice how it tells you what branch you are in.

Ans: git status

2. Use the branch command to create a new branch named my\_first\_branch.

Ans: git branch my\_first\_branch

3. Use the checkout command to switch to it.

Ans: git checkout my\_first\_branch

4. Make a couple of commits in the branch – perhaps adding a new file and/or editing existing ones.

Ans: echo "Hello from branch" > branch\_file.txt

git add branch\_file.txt

git commit -m "Add branch\_file.txt"

5. Use the log command to see the latest commits. The two you just made should be at the top of the list.

Ans: git log --oneline

6. Use the checkout command to switch back to the master/main branch. Run log again.

Ans: git checkout master

a. Notice your commits don’t show up now.

b. Check the files also – they should have their original contents.

Ans: git log --oneline

7. Use the checkout command to switch back to your branch.

Ans: git checkout my\_first\_branch

a. Use log --graph to take a look at the commit graph; notice it’s linear.

Ans: git log --graph

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.

Ans: git checkout master

git merge my\_first\_branch

a. Look for information about it having been a fast-forward merge.

Ans: output: Updating fb517c8..224c538

Fast-forward

b. Look at the git log, and see that there is no merge commit.

Ans: git log --oneline

c. Take a look at the commit graph and see how it is linear.

Ans: 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'

9. Switch back to your branch (my\_first\_branch). Make a couple more commits.

Ans:  git checkout my\_first\_branch

echo "New changes in branch" > branch\_new.txt

git add branch\_new.txt

git commit -m "Add branch\_new.txt"

echo "Edit branch file again" >> branch\_file.txt

git add branch\_file.txt

git commit -m "Another edit in branch\_file.txt"

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.

Ans: git checkout master

echo "Main branch edit" > main\_file.txt

git add main\_file.txt

git commit -m "Edit main branch with new file"

11. Now merge your branch again.

Ans: git merge my\_first\_branch

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.

Ans: git log --oneline

 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'

 git merge my\_first\_branch

git log --oneline --graph --all

**Exercise 3 : Undoing Changes in git**

**Main Task: Undoing Changes in Git**

1. From your main/master branch, create a new branch called ‘undoing\_changes’

Ans: git checkout master

git checkout -b 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.

Ans: Set-Content -Path file3.py -Value 'print("This is file3")'

git add file3.py

4. Use the commit command to commit the file.

Ans: git commit -m "Add file3.py"

5. Edit the contents of file3.py by adding a new line.

Ans: Add-Content -Path file3.py -Value 'print("New line in file3")'

a. Use the diff command to see the changes.

Ans: git diff

6. Use the git checkout command to discard the changes to file3.py.

Ans: git checkout -- file3.py

a. Use status to verify the changes have been undone.

Ans: git status

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

Ans: The changes I made in step 5 are gone

c. Could you have achieved the same result with a different command ? Hint: Check the message of git status

7. Create and commit a new file called file4.py.

Ans: Set-Content -Path file4.py -Value 'print("This is file4")'

git add file4.py

git commit -m "Add file4.py"

8. Use the git revert command to undo the commit you just made.

Ans: git revert HEAD

a. What does the revert command do compared to checkout?

Ans: discards uncommitted changes in the working directory.

b. Check the commit history using the log command.

Ans: git log --oneline

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.

Ans: git show HEAD

9. Make a new commit with changes in both file3.py and file4.py.

Ans: Add-Content -Path file3.py -Value 'print("Change in file3 for reset demo")'

Add-Content -Path file4.py -Value 'print("Change in file4 for reset demo")'

git add file3.py file4.py

git commit -m "Update file3 and file4 for reset demo"

10. Use git reset --soft HEAD^

Ans: git reset --soft HEAD^

a. Use the status command to check the changes.

Ans: git status

b. Can you describe what happened ?

Ans: The commit pointer (HEAD) moved back one commit, but the changes remain staged (in the index).

c. Do you still see your previous commit ?

Ans: No

d. What could you have done to avoid losing the commit after doing a reset ?

Ans: Create a branch before reset

11. Now, use git reset HEAD

Ans: git reset HEAD

a. Check the status again. What is the difference compared to the previous step ?

Ans: Now the changes are unstaged

b. Notice how we use HEAD instead of HEAD^ now ? Why are we doing that ?

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 ?

Ans: Because we only wanted to unstage files, not move back in history.

12. Finally, use git reset --hard

a. What happens to both the working directory and the commit history?

Ans: everything reset to the last commit.

b. We didn’t use any commit for this command, what do you think git used as default value ?

Ans: Git uses HEAD by default. That means the reset will revert everything to the last commit

c. What would have happened if you had done this command directly after step 9 ?

Ans: Discard the commit I just made

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

Ans: git log --oneline --graph

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

Ans:Set-Content -Path file5.py -Value 'print("This is file5")'

git add file5.py

git commit -m "Add file5.py"

Add-Content -Path file3.py -Value 'print("Another change in file3")'

git add file3.py

git commit -m "Update file3 again"

**Exercise 4 : Git rebase**

**Main Task**

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

a. Can you do it in one git command ?

Ans: git checkout -b feature-branch

2. In the feature-branch, create a file called feature1.py, write some content into it, and commit the file.

Ans: Add-Content -Path feature1.py -Value 'print("Another change in file3")'

3. Switch back to the main branch.

Ans: git checkout master

4. Create a new file on the main branch called main.py, add content to it, and commit the file.

Ans: echo 'print("Main branch code")' > main.py

git add main.py

git commit -m "Add main.py with main branch content"

5. Check the commit history before your next step: git log --oneline --graph --branches

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 ?

Ans: Rebase takes the commits from feature-branch and re-applies them on top of master.

The base branch is master

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

c. What has changed?

Ans: The history is now linear.

d. What’s the difference between this and merging main into feature-branch ?

Ans: Merge keeps history and creates a merge commit.

Rebase rewrites history so it looks like you always built on top of main

7. Rebase with Conflicts:

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

Ans: git checkout master

echo 'print("Add new things")'>feature1.py

git commit -m "Add new things to feature file"

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

Ans: git checkout feature-branch

echo 'print("Add new things in feature-branch")'>feature1.py

git commit -m "Add new things to feature file in feature branch"

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

Ans: git rebase master

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

Ans: output: feature1.py

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.

Ans: git add feature1.py

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?

Ans: feature-branch is unchanged.

main is unchanged.

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

Ans: git add feature1.py

git rebase --continue

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?

Ans: The history is linear.

No merge commits.

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

Ans: The feature-branch commits were rebased on top of main, replayed one by one

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

Ans: A merge would create a branching graph and a merge commit.

A rebase makes it look like your branch was always based on the latest main.

10. Create a few more commits on both main and feature-branch.

Ans: git checkout master

echo "print('Another change in master')" >> main.py

git commit -am "Update main.py"

git checkout feature-branch

echo "print('Another feature improvement')" >> feature1.py

git commit -am "Update feature1.py again"

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?

Ans: Squashing combines multiple commits into one.

c. Modify a commit during an interactive rebase.

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

Ans: git add feature1.py

git commit --amend

git rebase --continue

12. Use git log --oneline --graph --branches to inspect the commit history and see how the rebase has altered it.

Ans: Graph remains linear, because rebase rewrites history, unlike merge which creates merge commits