BYOD 2 GIT

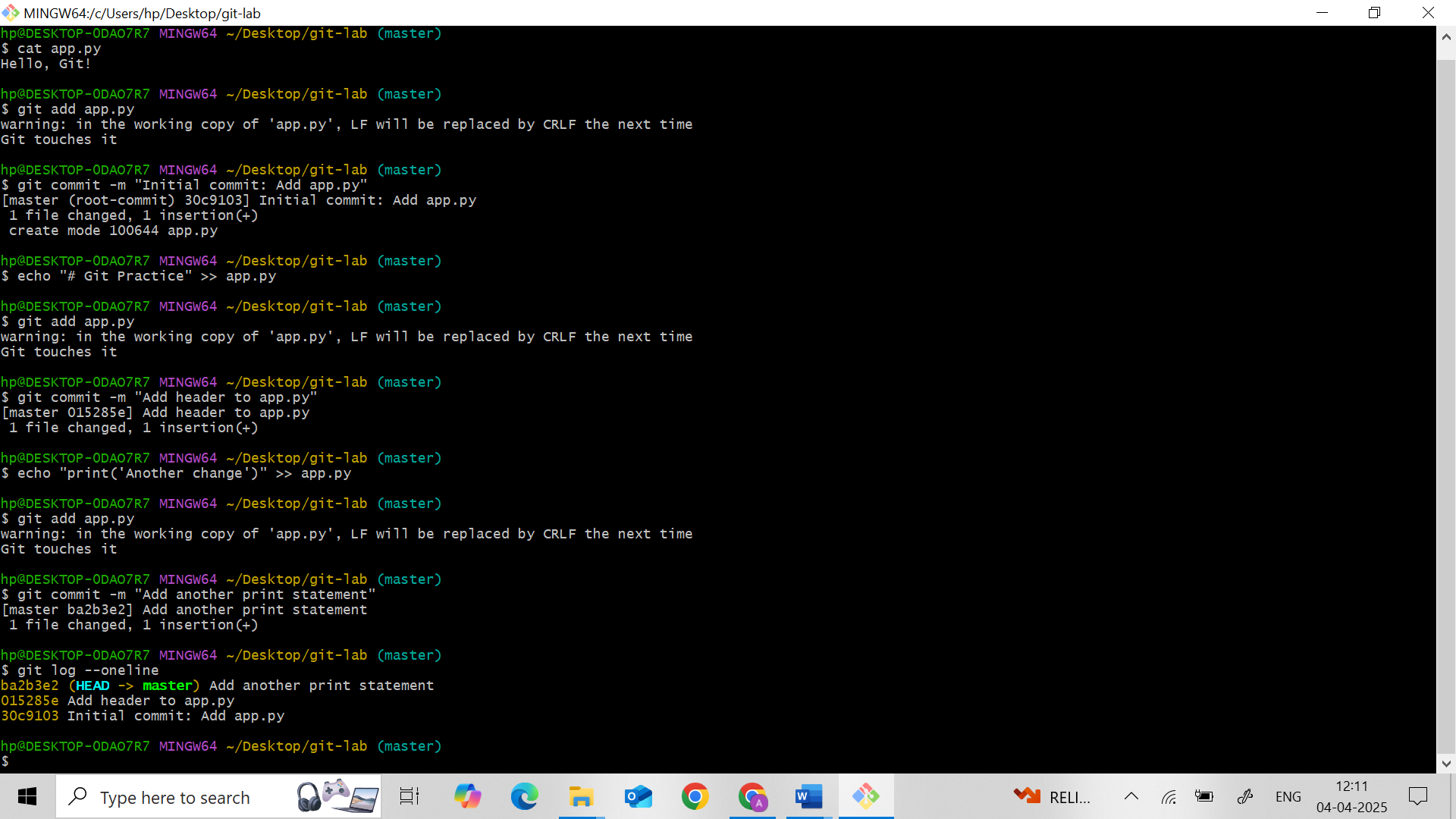
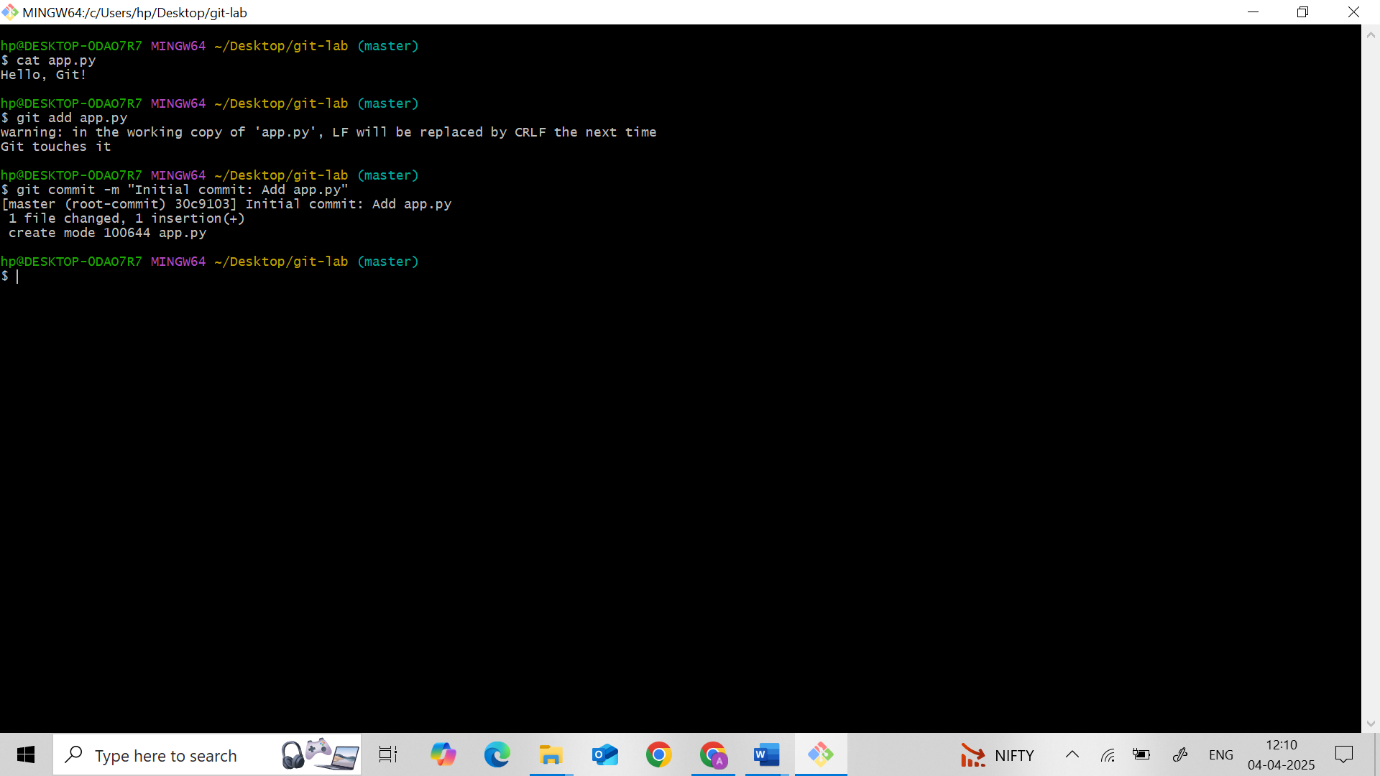
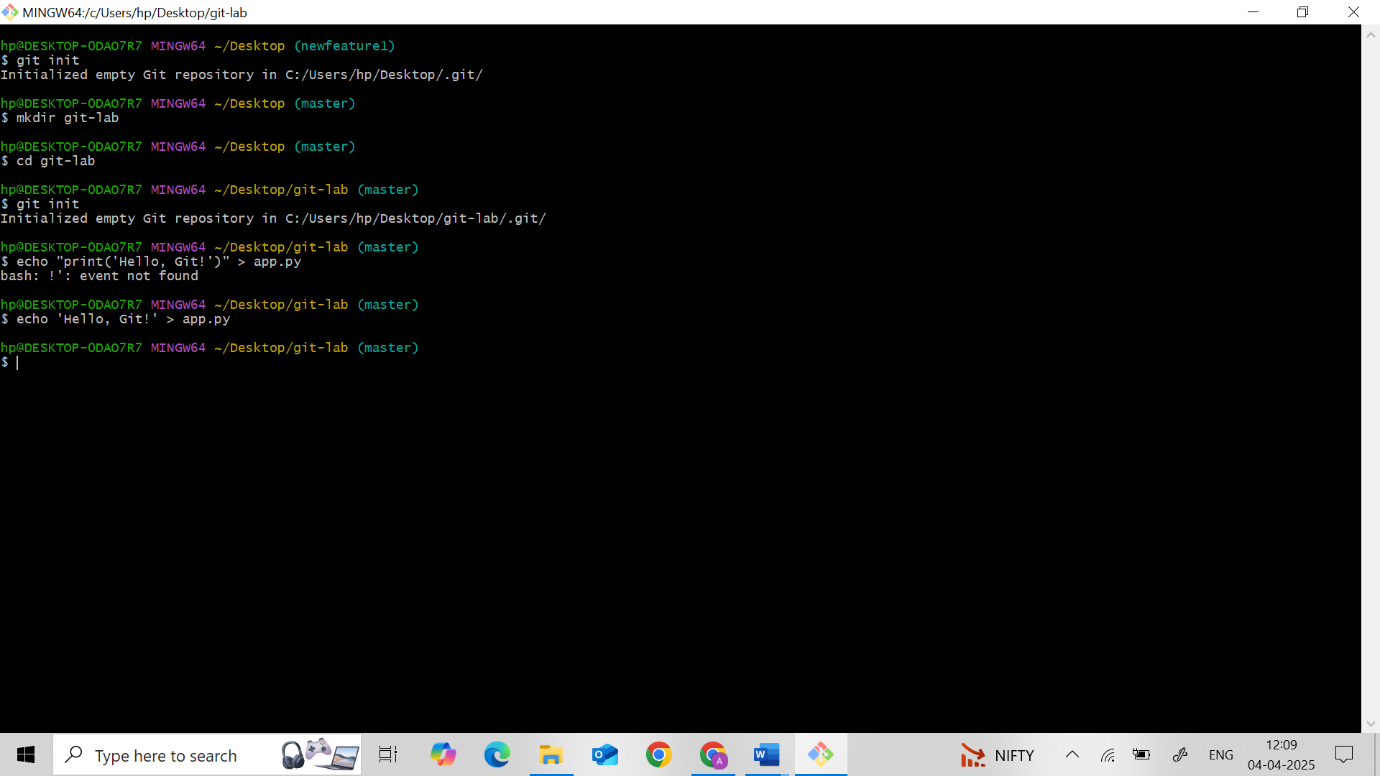
Name: Aditya Kumar

Reg no. 12301713

Section k23xr

**Part A: Setup and Initial Commits**

1. **Repository Setup:**
   * Create a new directory for the lab.
   * Initialize a new Git repository.
   * Create an initial file (e.g., app.py or index.html) with some content.
   * Stage and commit your changes.
2. **Build the Commit History:**
   * Make at least two additional changes to your file(s) and commit each change.
   * Verify that your commit history contains multiple commits.

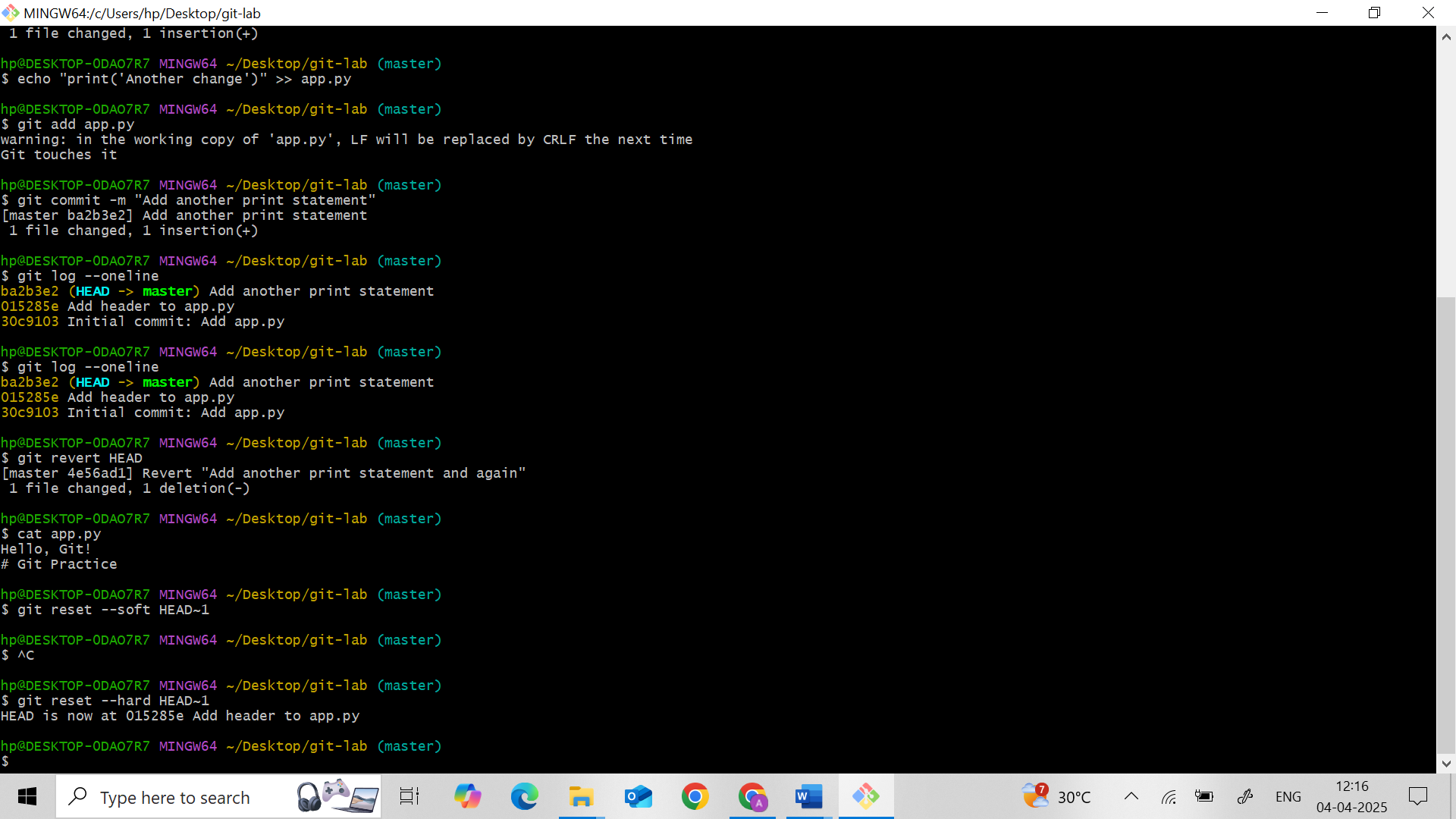


**Part B: Rollbacks in Git**

1. **Explore Commit History:**
   * Use a Git command to view your commit history and identify commit hashes.
2. **Reverting Changes:**
   * Use a Git command to revert the latest commit by creating a new inverse commit.
   * Verify the changes in your file to see the effect of the revert.
3. **Resetting the Repository:**
   * Practice using both soft and hard resets to move the repository state backward.
   * Observe and document the differences in how each reset type affects the working directory and staging area.

**Soft Reset (undo commit, keep changes staged)**

**Hard Reset (undo commit and discard changes)**



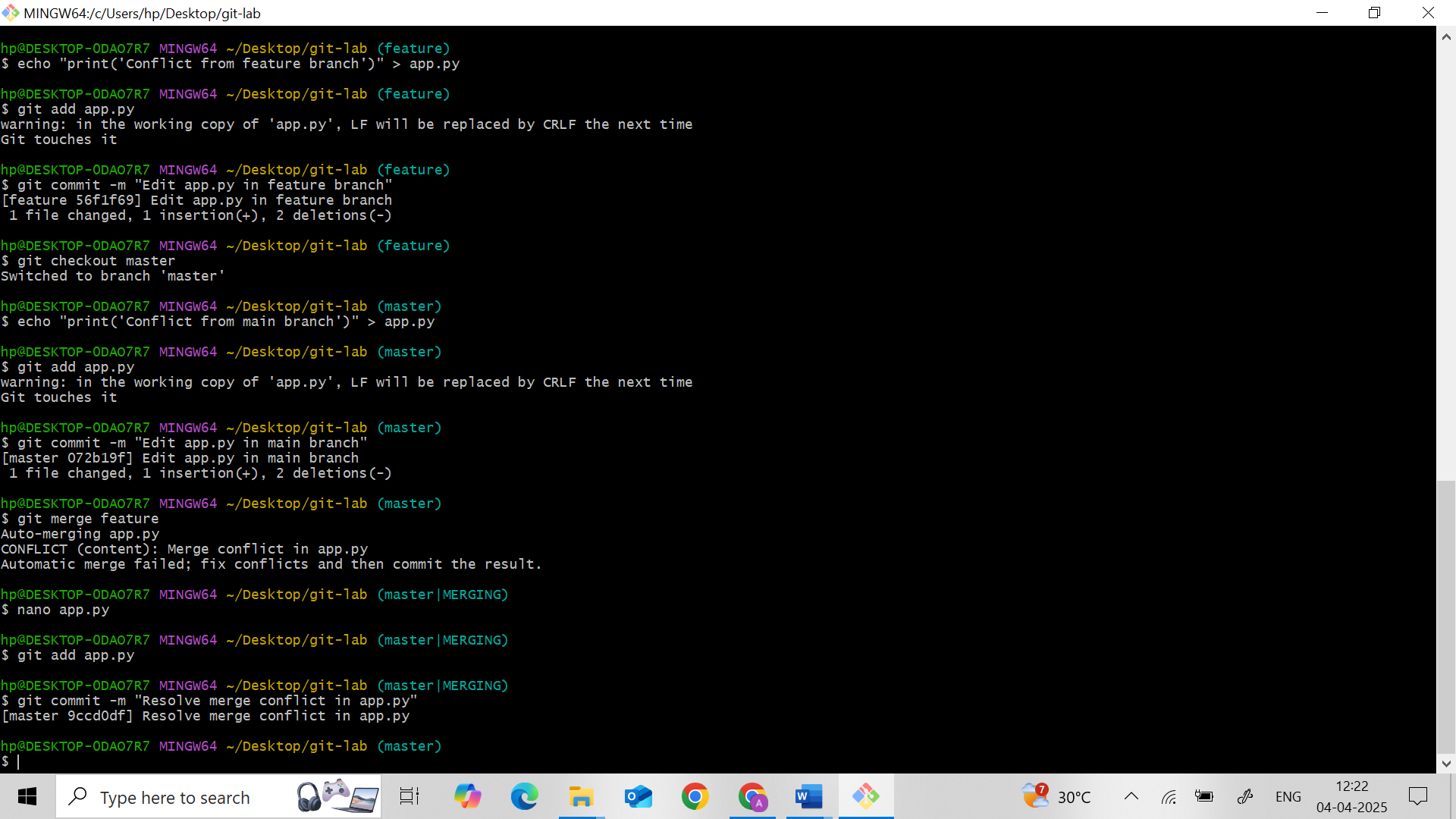
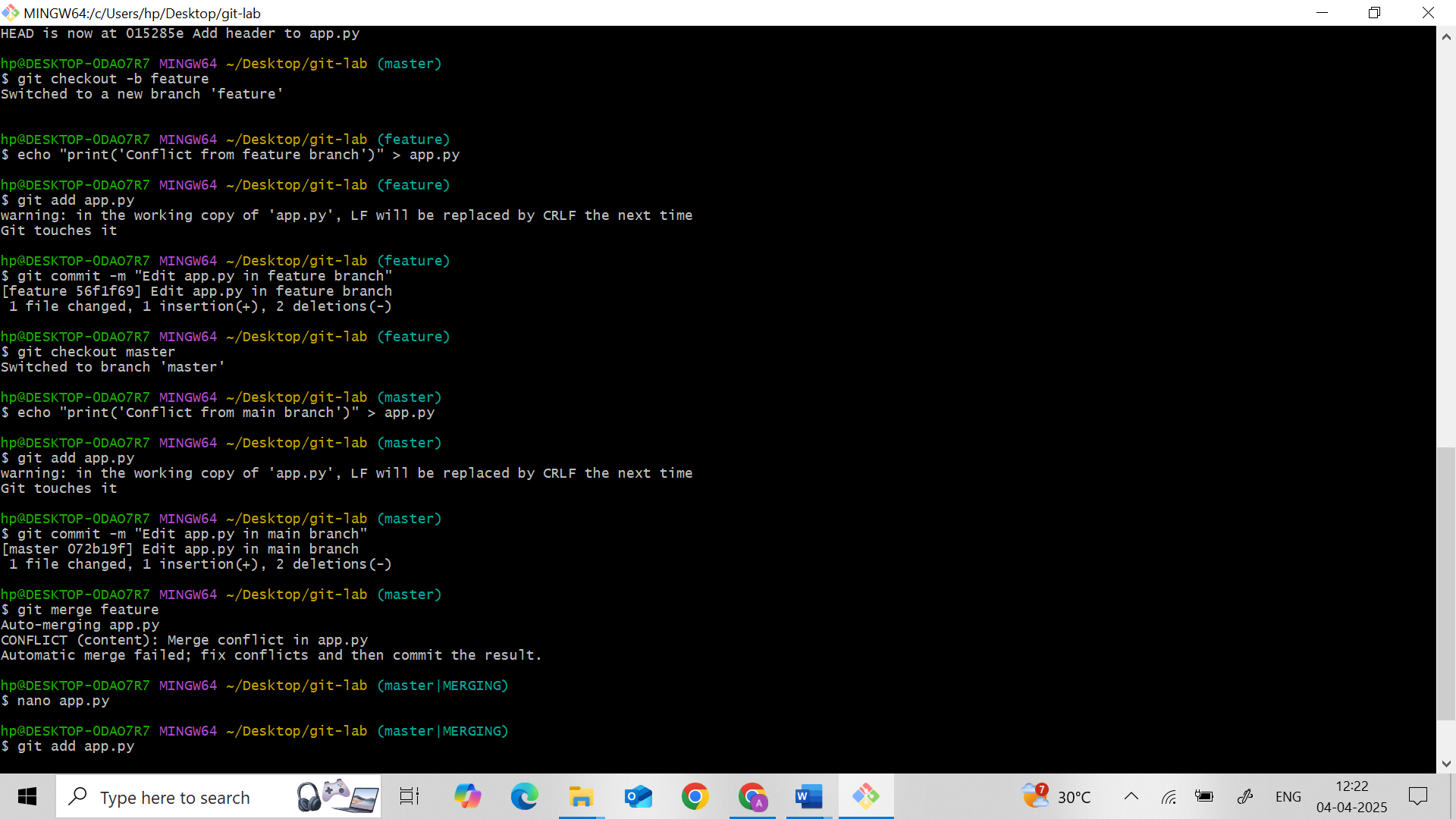
1. **Reflection:**

Briefly describe the role of HEAD in Git and how it changes during resets

**HEAD** is a pointer to the current branch reference.  
During soft reset, HEAD moves but your changes stay staged.  
During hard reset, HEAD moves and resets everything — staged and working directory.

**Part C: Merge Conflict Resolution**

1. **Branch Creation and Divergence:**
   * Create a new branch to simulate parallel development.
   * Modify the same section of a file differently on this new branch and commit the change.
   * Switch back to the main branch, make a different modification to the same section, and commit.
2. **Merging and Conflict:**
   * Attempt to merge the new branch into the main branch.
   * Encounter and identify the merge conflict.
3. **Conflict Resolution:**
   * Manually resolve the merge conflict by editing the conflicting file.
   * Stage and commit the resolution.
4. **Reflection:**
   * Explain how the merge conflict occurred and outline the steps you took to resolve it.



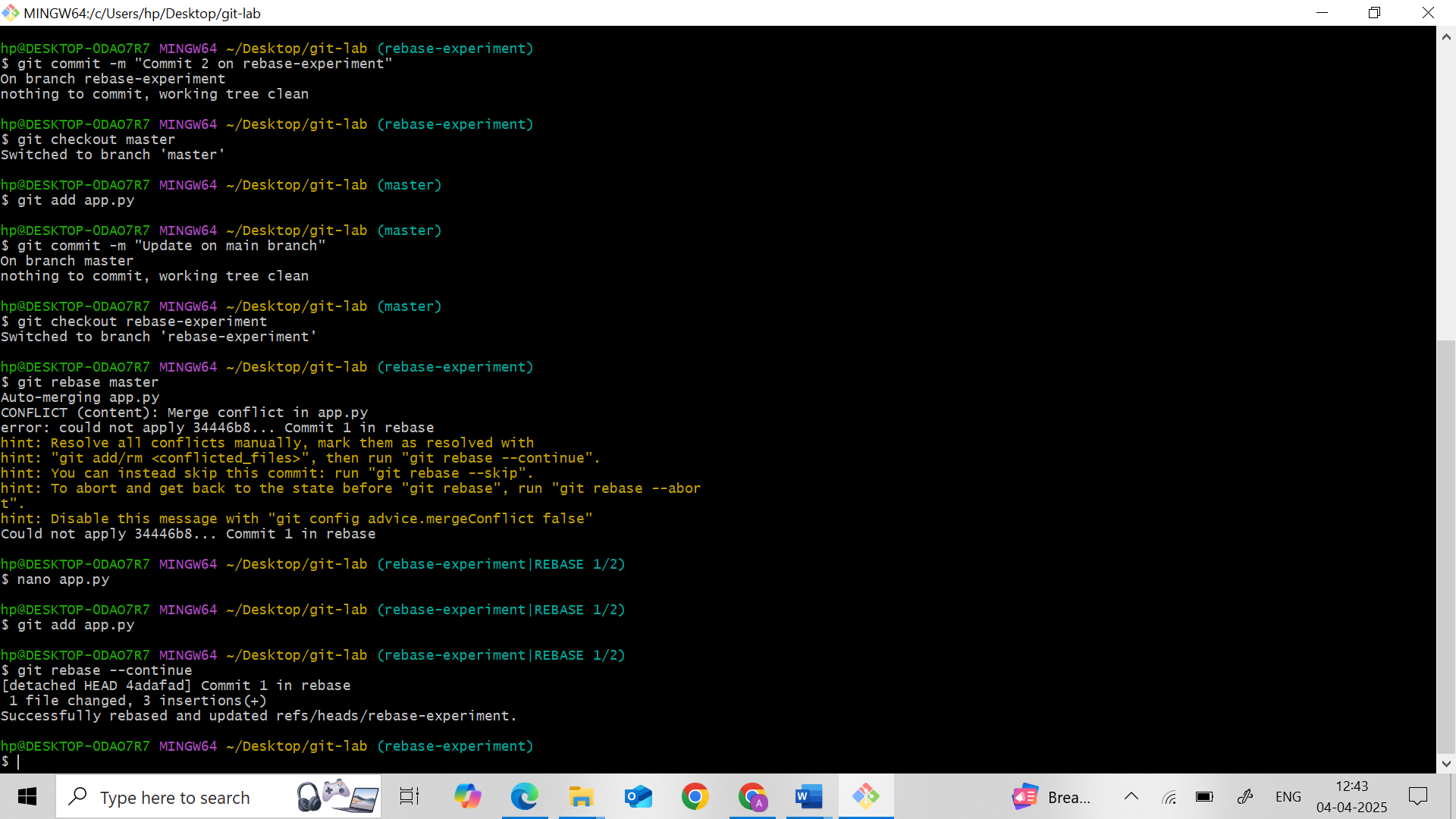
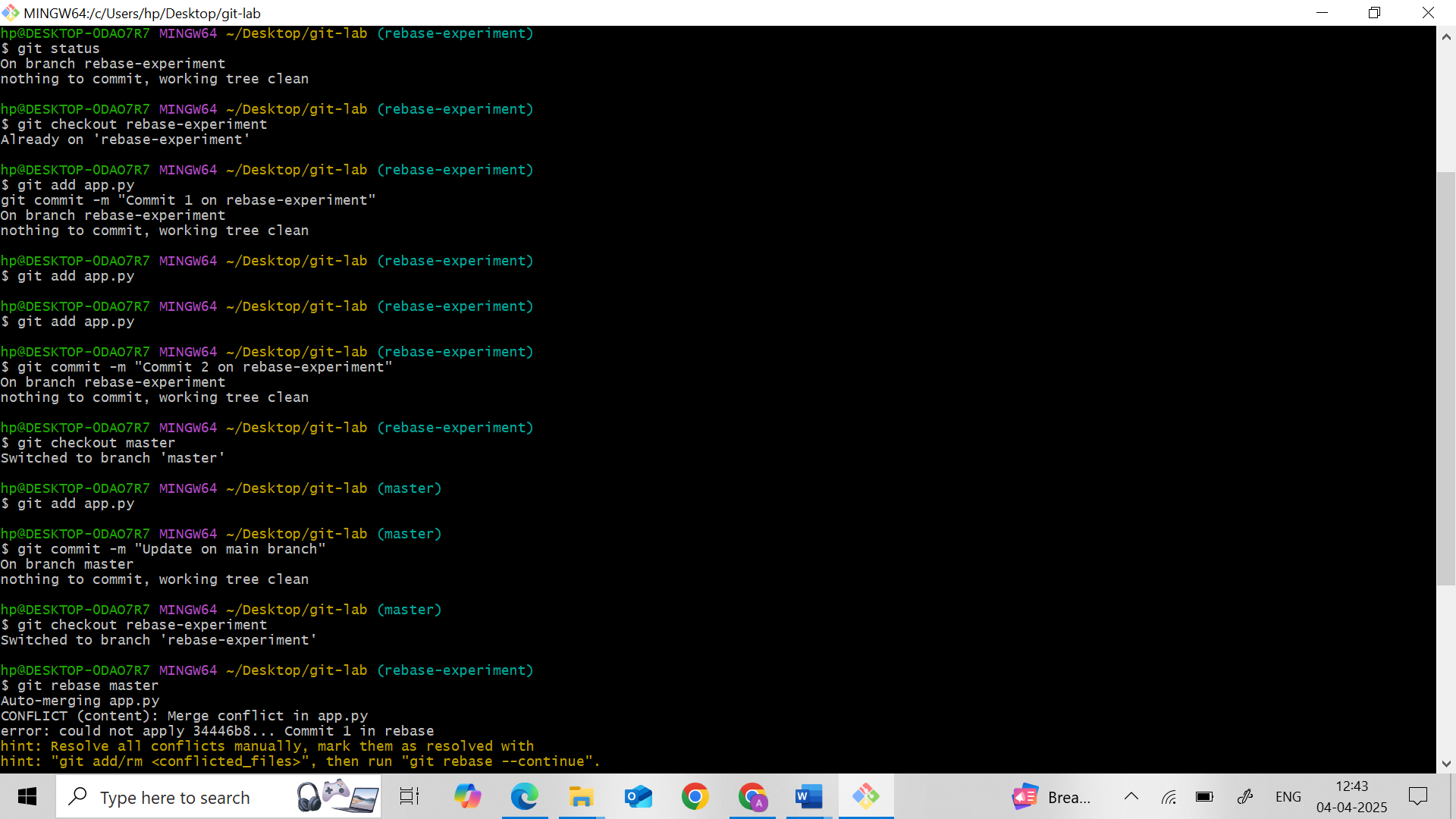
**4. Reflection**

Merge conflict occurred because both branches changed the same line.  
To resolve, we manually chose which content to keep in the file, then staged and committed the fix.

**Part D: Rebasing vs. Merging**

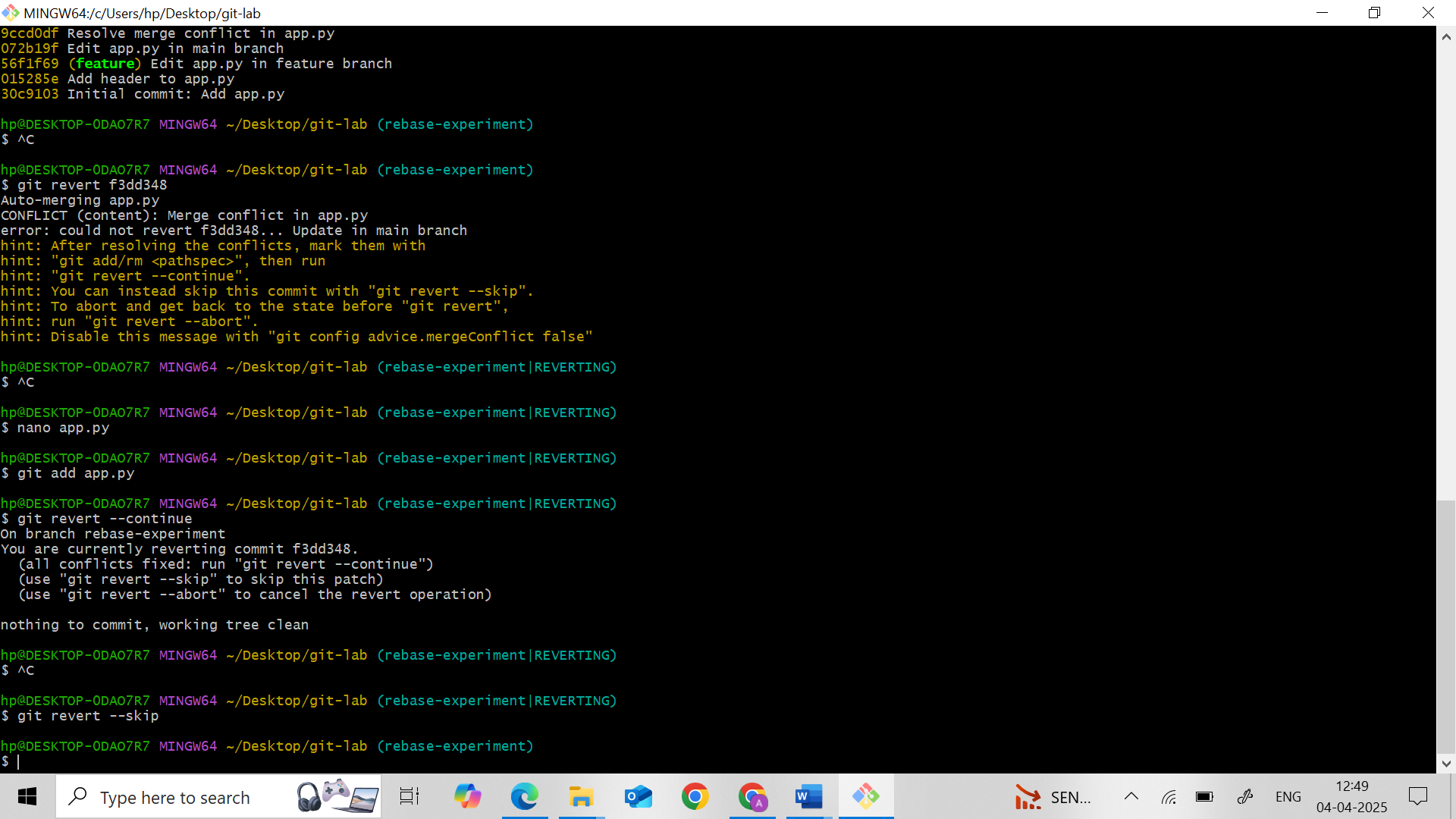
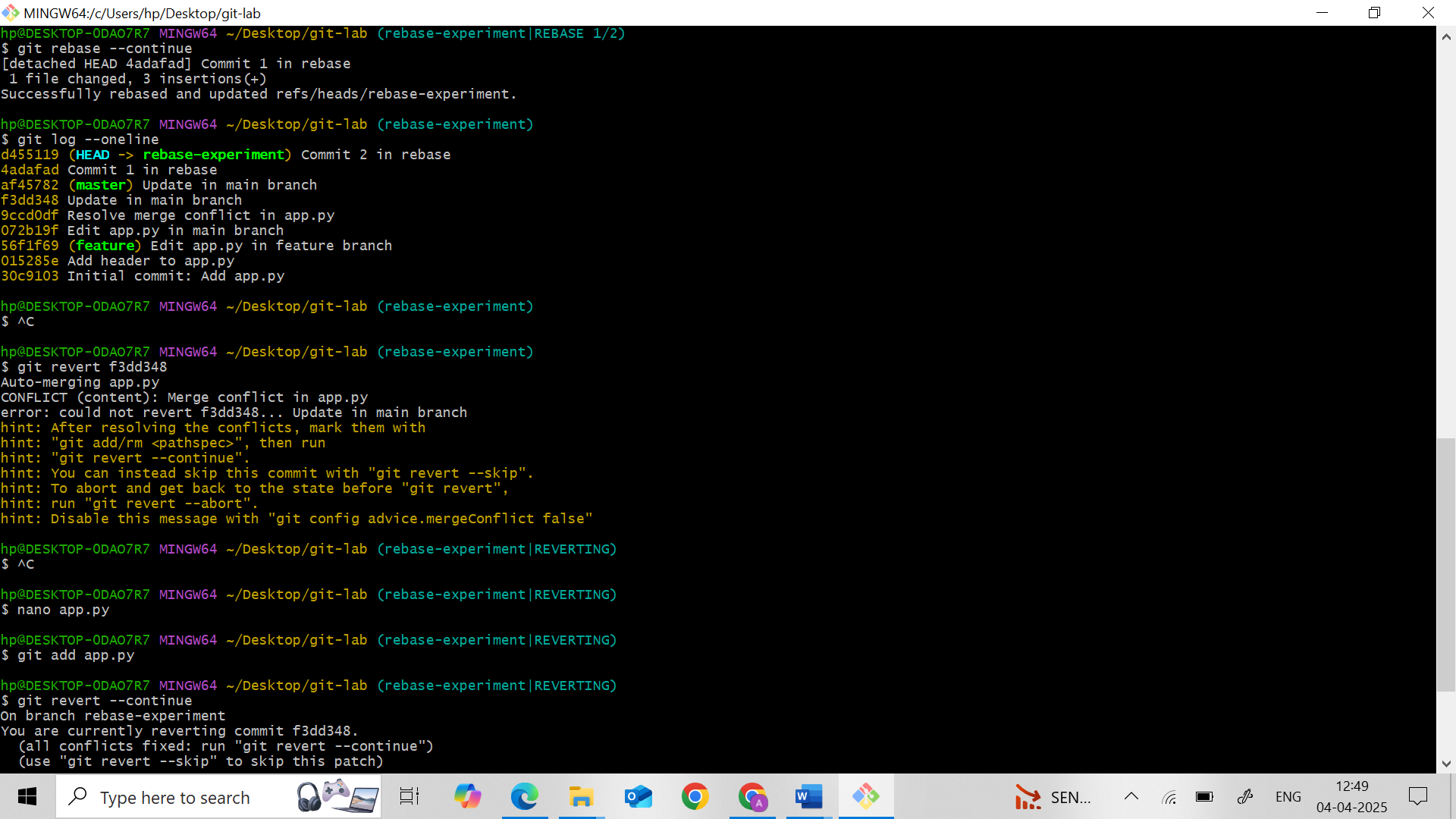
1. **Branch Experimentation:**
   * Create another branch for experimenting with rebasing.
   * Make a couple of commits on this branch.
2. **Main Branch Update:**
   * Switch back to the main branch and make an update commit.
3. **Perform a Rebase:**
   * Rebase the experimental branch onto the updated main branch.
   * Resolve any conflicts that arise during the rebase process.
4. **Comparison:**
   * Reflect on the differences between merging and rebasing in terms of commit history and workflow.
   * Document the advantages and potential pitfalls of each method.

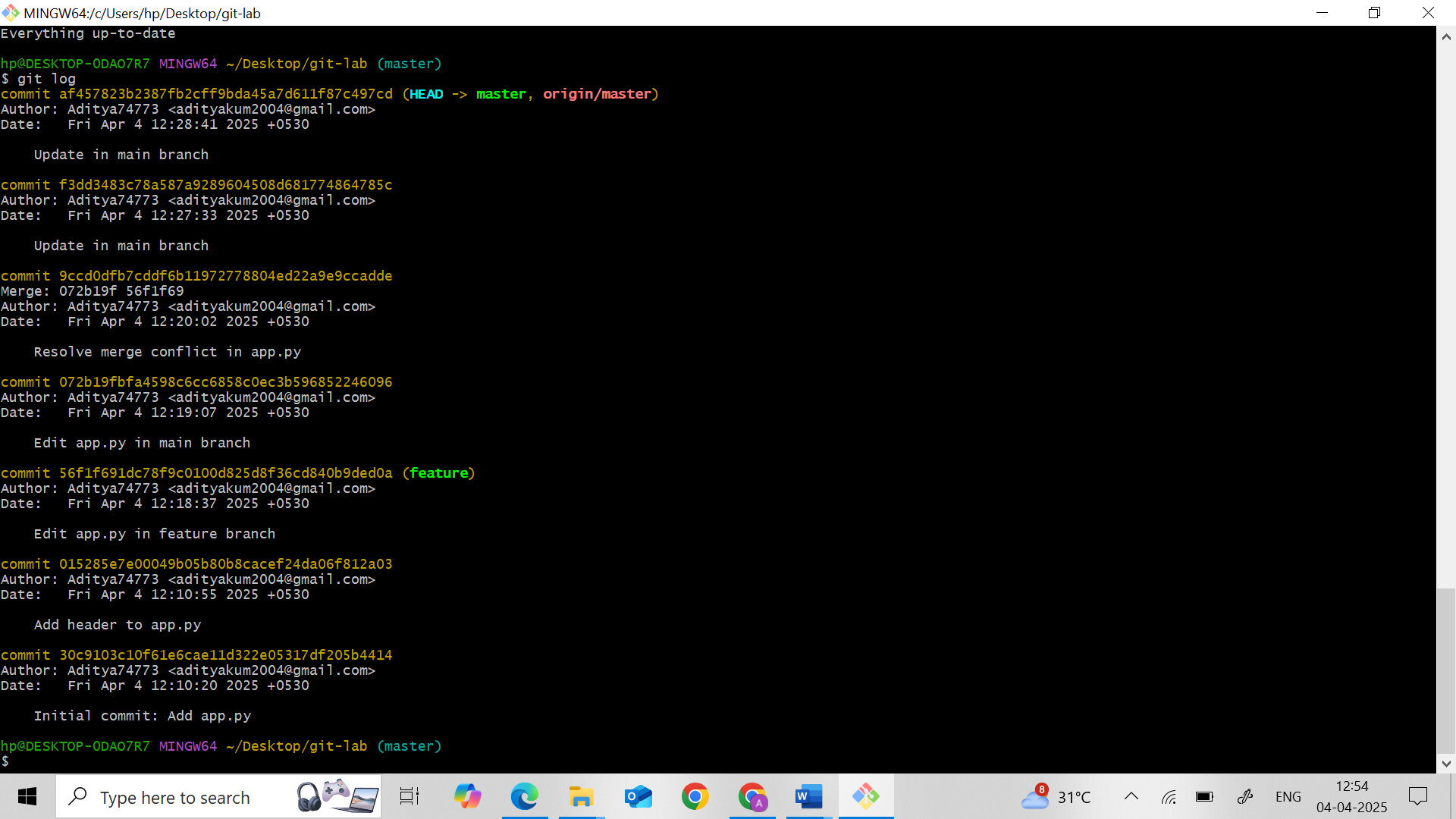
* **Merging** keeps both branch histories and adds a merge commit.
* **Rebasing** rewrites history to make it linear — cleaner but riskier.
* **📘 Comparison: Merging vs Rebasing in Git**
* **🔀 Merging**
* **Command:** git merge branch-name
* **✅ Advantages:**
* **Preserves complete history**: Shows exactly when and how branches diverged and merged.
* **Safe and non-destructive**: Existing commits remain unchanged.
* **Easy to understand in team workflows**.
* **⚠️ Pitfalls:**
* **Commit history can get messy** with lots of merge commits.
* Harder to follow a linear progression of features and changes.
* **🔁 Rebasing**
* **Command:** git rebase branch-name
* **✅ Advantages:**
* **Creates a clean, linear history** (looks like all work happened sequentially).
* Easier to **navigate with commands** like git log or git bisect.
* Preferred for feature branch updates **before merging into main**.
* **⚠️ Pitfalls:**
* **Rewrites commit history** (can cause confusion or errors if already pushed).
* **Dangerous on shared branches** – never rebase public branches that others are using.
* Conflict resolution can be more complex if there are many commits.



**Part E: Reverting a Specific Commit**

1. **Commit Identification:**
   * Identify a specific commit from your history that you want to undo.
2. **Reverting:**
   * Use Git to revert that specific commit.
   * Verify that a new commit has been created which undoes the changes introduced by the original commit.
3. **Reflection:**
   * Summarize how reverting a commit differs from resetting the repository.





* git revert = **undo with a new commit** (safe)
* git reset = **move HEAD and optionally delete commits** (powerful but risky)

**Deliverables**

* **Git Repository:** Ensure your repository includes:
  + A series of commits reflecting your initial work, rollback attempts, merge conflict resolution, rebasing activity, and the specific revert.
* **Documentation:** Provide a brief written summary that covers:
  + How HEAD moves during the various rollback operations.
  + Your approach to resolving merge conflicts.
  + A comparison of rebasing versus merging and their impacts on project history.

**Documentation Summary**

**How HEAD moves during rollback operations:**  
In Git, HEAD points to the current commit in the checked-out branch. During a git revert, a new commit is created that undoes the changes of a specific commit, and HEAD moves forward to that new commit. During git reset, the HEAD is moved to a previous commit:

* In git reset --soft, HEAD moves but the working directory and staging area remain unchanged.
* In git reset --mixed, HEAD moves and the staging area is cleared.
* In git reset --hard, HEAD moves and both the working directory and staging area are reset, discarding all changes.

**Approach to resolving merge conflicts:**  
To resolve a merge conflict, changes were made to the same part of the file on two branches. Upon merging, Git flagged the conflict. The file was opened, and the conflicting sections marked by Git were manually edited to keep the correct content. After saving the changes, the file was staged using git add, and the merge was completed using git commit.

**Comparison of rebasing versus merging and their impacts on project history:**  
Merging creates a new merge commit and retains the history of both branches, showing a true representation of how changes were developed in parallel. It’s safer for collaborative environments.  
Rebasing, on the other hand, re-applies commits from one branch onto another, creating a linear commit history. This makes the history cleaner but can be risky if conflicts arise or if the branch has already been shared. Rebasing rewrites commit history, so it’s best used for local, private branches before pushing.