Project 1 - Basics of Version Control

Learning Objectives:

- Collaborate with a student partner using **git command**s.
- Resolve **merge conflicts** locally.

Project Features:

- Hands-on experience with the following version control practices:
  - Commit
  - Branch
  - Stash
  - Pull Request
  - Merge Conflicts
  - Tag
  - Revert
- Instructions on how to use GitHub collaboratively, including creating accounts, using templates, SSH keys, and setting branch protection rules.

Business Context

What is the general business problem you'll solve?	Why is it important to the business to solve?	What is the solution and how could the solution be convincingly presented?
Your company is transitioning to using git as the primary version control system to manage code changes efficiently. As a Software Engineer, you are not only responsible for developing high-quality software but also for adopting best practices in version control to improve code management. The company has faced challenges with code versioning in the past, leading to instances where software contributions were overwritten or teams worked on outdated code versions, causing significant setbacks in product development and delivery. This mismanagement sometimes resulted in broken features in the final product, leading to customer dissatisfaction. To address these challenges, your company has decided to enforce a structured version control system using git. Your task is to understand and implement advanced git practices in a hands-on project building a Python task tracking app. You'll be using git to manage changes, handle conflicts, and make sure that the codebase remains stable and up-to-date.	Engineer Productivity: Frequent need to have discussions with other teams over merge conflicts or data structures results in lost time writing solutions. With effective git practices, developers can focus on their core tasks at hand. Reduced Annoyance and Frustration: The tensions between teams and other engineers can arise when they have to move delivery times due to code conflicts. By enforcing good practices such as working on separate modules and pulling changes frequently, engineers can avoid conflicts. Reduction in Time for Recovery: Proper version control allows for faster debugging if software was reported to fail. Reverting to previous code and pinpointing what caused the issue helps teams release their products sooner for the consumer to enjoy the product.	Engineering leadership has tasked every developer across the company with completing a self-guided series of tasks in pairs with another engineer from a different team. The self-guided tasks involve you working with a partner on using git best practices. You will be expected to work alongside each other and communicate effectively when problems arise. To complete this, there are 6 tasks below and some tasks contain multiple steps within each: ● Committing regularly (committing at every significant change, such as when you complete a single task and before you move on to the next one) ● Branching (separating state at any point to ensure that you can raise Pull Requests) ● Resolving merge conflicts (using simple text editor and terminal to resolve) ● Reverting (going to the previous version of the code) ● Tagging (saving state to serve as milestone or release)

Overview of Starter Material

- A GitHub template repository that students can start from
- Instructions on how to setup a branch protection rule
- Instructions with what part of code to change / fix for each student (separate for "Student A" and "Student B")
- Instructions on how to implement an algorithm for merge conflict (separate for "Student A" and "Student B")
- Students will need to set up their own GitHub accounts and install appropriate tools (IDE like VSCode, Git Bash, WSL, etc.)

Project Requirements

Languages / Technologies:

- Git
- GitHub
- Python

Team Size:

- Two people on one team (Student A and Student B)

Suggested Project Timeline:

- 1-2 weeks

Grading / Deliverables

You will be evaluated against 7 deliverables as described in the rubric below:

- Successful git and remote repository setup
- Successful commit with some code changes to the **main** branch and **pull** to receive partner's changes (tricky part: the partner will need to do a **git pull** before pushing)
- Feature to the codebase delivered via a separate **branch** and a successful merge via a **pull request**
- Successful use of **git stash** to temporarily store changes that are not ready to be committed
- Successful resolution of a **merge conflict** due to changes introduced by partner
- Successful **revert** of a commit that introduced breaking changes
- Successful use of **git tag** to create tags

Requirements	**Deliverable(s)**	**0 – Incomplete / Missing**	**1 – Does not satisfy requirements**	**2 - Satisfies Requirements**	**3 – Exceeds Requirements**

1. Successful git and remote repository setup
(Task 0 below)
	Successfully set up a local Git repository.
	Successfully set up a remote repository on a platform like GitHub.
	Provide screenshots or documentation demonstrating the local and remote repository setup.
	Demonstrate understanding of basic Git commands (e.g., init, clone).
	Include a brief explanation of the Git commands used.
		No evidence of local or remote repository setup.	Setup is incomplete or incorrect.
			The remote repository may have been set up, but the cloned repository is not in sync or incorrect.
			For example: setup with **git init** instead of **git clone** and not connected to remote; no initial code present
				Local and remote repositories are set up correctly using **SSH keys**.	Local and remote repositories are set up correctly plus proper use of **git config**
					$ git config --list
					user.email=name@collegeedu
					user.name=First Last
					$ git remote show origin
					* remote origin
					Fetch URL: git@github.com:username/your-repository-name.git
					Push URL: git@github.com:userna

					me/your-repository-name.git $ git log Shows their name and email
2. A successful commit with some code changes to the **main** branch and **pull** to receive partner's changes (Task 1 below)	Demonstrate understanding of basic Git commands (e.g. remote, push, pull). Screenshots of the local **git log** showing the commit. Screenshots of the GitHub commit history. Documentation explaining the need for a **git pull** before pushing.	No evidence of commit or pull in local or remote **git log** shows no commits and GitHub UI shows no commits from student	Commit or pull is incomplete or incorrect **git log** shows commit present in local repository, but no commit present in remote GitHub repository (missing **git push origin**) **git log** shows no commit from Student B student present in Student A's local	Successful commit and pull demonstrated. **git log** shows Student A's and Student B's commits. GitHub UI shows both commits present	Includes advanced Git commands or collaboration strategies. Such as, but not limited to: A descriptive commit message explaining the change made Use of a branch and pull request before the next stage of the assignment
3. Feature to the codebase delivered via a separate **branch** and a successful merge via a **pull request** (Task 2 below)	Create a feature branch for a specific task or bug fix. Implement changes in the feature branch. Merge the feature branch into the main branch via a pull request. Screenshots of the local **git log** and **git branch -a** showing the feature branch and changes. Screenshots of the GitHub pull request and merge process. Explanation of the purpose of the feature branch.	No evidence of a feature branch or successful merge.	Feature branch or merge process is incomplete or incorrect Didn't create a branch, pushed directly to main, didn't raise a PR	Successful feature branch and merge demonstrated.	Includes advanced branching strategies or collaborative practices Such as, but not limited to: A squash and merge in case of multiple commits Set up a branch protection rule to stop anyone from pushing to main
4. Effectively utilize git stash to temporarily store changes that are not ready to be committed (Task 3 below)	Screenshots of using git stash to save uncommitted changes. Screenshots of using **git stash pop** from git stash to apply stashed changes back into the working directory. 1-2 sentences with a brief explanation of when and	No evidence of using or understanding git stash.	Demonstrates basic use of git stash, but with errors or misunderstanding of its purpose.	Successfully uses git stash to save and reapply changes, with a clear understanding of its purpose.	Demonstrates advanced use of git stash, such as managing multiple stashes or using stash pop and stash drop. Includes thoughtful insights on when to use stashing in a collaborative project.

	why to use git stash.				
5. Successful resolution of a merge conflict due to changes introduced by partner (Task 4 below)	Intentionally introduce a merge conflict. Demonstrate understanding of merge conflict causes. Utilize appropriate tools or commands to resolve conflicts (manual resolution through IDE and terminal preferred). Screenshots or logs showing the intentional merge conflict. Screenshots of the resolved conflict. Brief documentation on the resolution process.	No evidence of a merge conflict or resolution.	Conflict resolution is incomplete or incorrect Such as: A merge conflict is a mix of both code versions instead of single correct version	Successful conflict resolution demonstrated both in local and remote through a pull request	Includes insights into preventing conflicts or advanced conflict resolution strategies Such as, but not limited to: Brief documentation talking about branch naming approach Proper code structure, such as code modularization Frequent rebase and pull Explain the difference between two methods: rebase and merge
6. Successful revert of a commit that introduced breaking changes (Task 5 below)	Intentionally introduce a commit with breaking changes. Demonstrate understanding of the **git revert** command. Successfully revert the commit. Screenshots or logs showing the breaking commit. Screenshots of the successful revert. Brief documentation on the revert process.	No evidence of a breaking commit or successful revert.	The revert process is incomplete or incorrect.	Successful commit revert demonstrated.	Includes insights into when to use revert and potential challenges. Explains the difference between revert and reset and the implications of each
7. Effective tags for significant points in the project's history, like version releases. (Task 6 below)	Screenshots of: Creation of annotated and/or lightweight tags in your local Git repository. Push the tag to the remote GitHub repository. Checking out the specific tag locally. Two sentences with a brief explanation of tags and the difference	No evidence of using or understanding git tag.	Demonstrates basic use of git tag, but with errors or misunderstanding of its purpose.	Successfully uses git tag for marking significant points in the project, understands the difference between annotated and lightweight tags.	Demonstrates advanced use of tags, such as using tags in release management, integrating tags with GitHub releases, or detailing the strategic importance of tagging in version control. Insightful explanation of best practices in tagging.

	between annotated and lightweight tags.				

Student Resources and Notes to Get Started

- Official Git Documentation
- GitHub Skills
- Git Basics Tutorial
- Interactive Git Branching Tutorial
- Git and GitHub for Beginners - Crash Course

Specific Topic resources:

- Using git stash
- Using git tag
- Resolving merge conflicts
- Creating pull requests
- Branch protection rules

Student Instructions

Version control is an essential part to any software engineering product and tools like `git` are used daily by engineers to deliver code.

Before you start this assignment, please determine with your partner who is going to be **Student A** and **Student B**. You may find the screenshots useful, where Student A is Tom Hanks and Student B is Tim Allen. There is no difference in workload. This is only used to determine who will perform what tasks to facilitate collaboration.

This assignment consists of multiple tasks each with various steps and requirements. You are expected to have a student partner to pass the assignment successfully. The tasks are meant to familiarize you and your partner with the following version control concepts:

- Commit
- Branch
- Stash
- Pull Request
- Merge Conflict
- Tag
- Revert

Task 0 - Setup your local environment, 5 Steps

Before you can make commits and contribute different versions of your code, you must set up your environment.

Step 1 - Install git

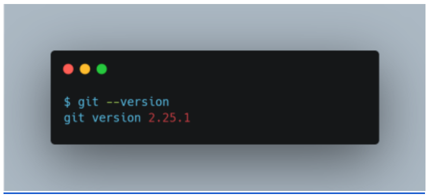
If you are a Windows user, you need to install a Unix environment to run git commands.

Follow these instructions to setup **Git Bash** or install it directly from the source.

Once you finish the installation, you can open Git Bash and make sure that it works by running **git --version**.

If you are a MacOs or Linux user, you already have git installed in your environment.

Make sure that is the case by running **git --version**.



```
$ git --version
git version 2.25.1
```

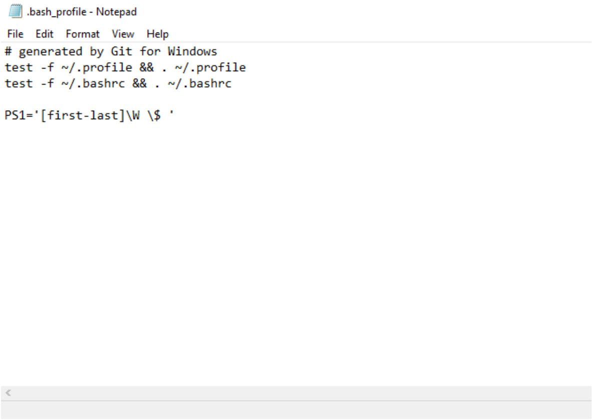
Take a screenshot of running the command in your terminal.

Step 2 - Change Bash prompt

As this is paired work, you must modify the terminal to show your full and last name so that your screenshots are easy to understand.

If you are using **Git Bash on Windows**, run the command: **notepad ~/.bash_profile**

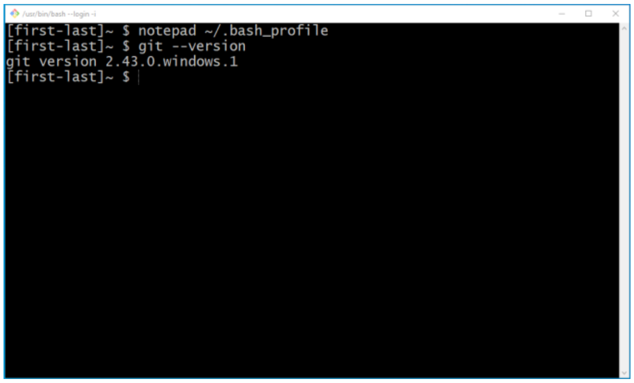
Which will then open Notepad letting you add the line **PS1='[first-last]\W \$ '** (make sure to substitute first and last with your full name):



```
# generated by Git for Windows
test -f ~/.profile && . ~/.profile
test -f ~/.bashrc && . ~/.bashrc

PS1='[first-last]\W \$ '
```

Remember that you need to add the last line yourself. You don’t have to modify the first lines.

Save and close. Then, restart Git Bash for the changes to take place.

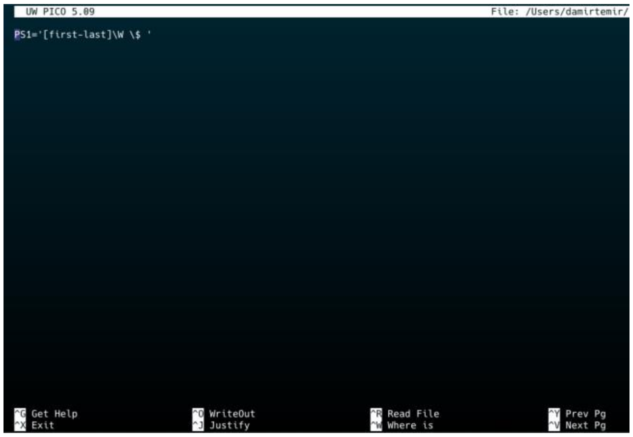


```
[first-last]~ $ notepad ~/.bash_profile
[first-last]~ $ git --version
git version 2.43.0.windows.1
[first-last]~ $
```

Note the first-last. You must add your name. For example, john-doe

If you are on WSL2, Linux, or MacOS, you need to modify a file named **.bashrc** by copying the same line **PS1='[first-last]\W \$ ':**

The simplest way to do so is with **nano** (if you have any contents in that file already, make sure to scroll down to add the line): **nano ~/.bashrc**

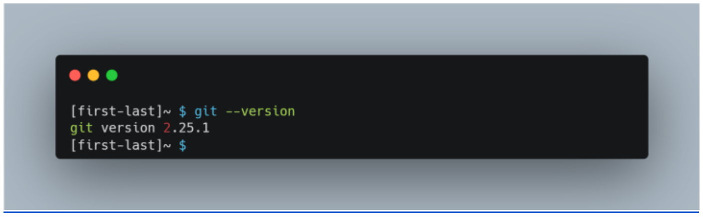


```
PS1='[first-last]\W \$ '
```

Your .bashrc may already have some configuration setup, make sure to scroll down by using ARROW down and only pay attention to the last line PS1

To save, press CTRL + X and then Y to save the edited file.

Then, restart the terminal.



```
[first-last]~ $ git --version
git version 2.25.1
[first-last]~ $
```

Step 3 - Create GitHub account

Sign up for a github.com account at https://github.com/.

Copy the URL link that leads to your account. Ex: github.com/yourusername

Step 4 - Setup SSH authentication

To successfully communicate with GitHub from your terminal, you must have a secure connection.

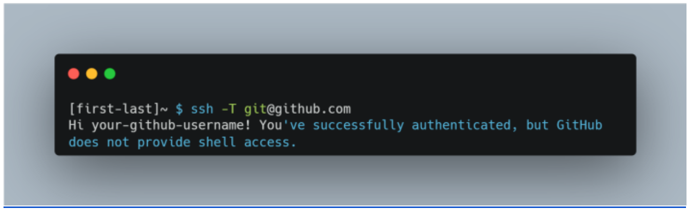
The secure connection is facilitated with the use of an **SSH** (Secure Shell Protocol) key that is kept privately on your machine.

Under no circumstances should you share the private SSH key with anyone. It is completely fine to share the public SSH key (ends with **.pub)** and that is what you will need to provide GitHub so that they know the communication is secure. Here is a small article explaining why and what is the difference between public and private keys.

You may follow these instructions to create a key and provide GitHub with the public key:

- Generating a new SSH Key and Adding it to the SSH Agent
- Adding a New SSH Key to Your GitHub Account

Make sure you can communicate with GitHub by running:



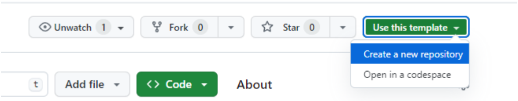
```
[first-last]~ $ ssh -T git@github.com
Hi your-github-username! You've successfully authenticated, but GitHub
does not provide shell access.
```

Take a screenshot of the above in your terminal with your GitHub username listed.

Step 5 - Clone repository

Student A: Navigate to the provided link and use the template to create your own repository.

Commented [MOU1]: Which one?



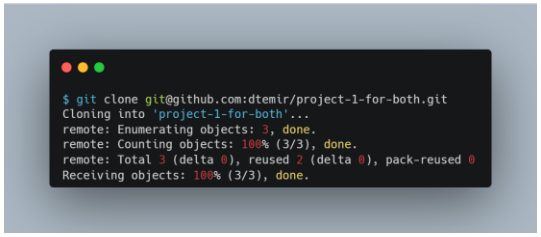
The next page will ask you to name your repository and choose from public or private. **Make sure to choose private**. Here are the docs to help.

The repository will be generated for you and you now need to invite **Student B and your course instructors** to your repository.

You need to navigate to **Settings -> Collaborators -> Add people**. There, you can provide the usernames of everyone you need to invite. Here are the docs to help.

Student B: You will receive an email with an invite from Student A. Make sure to accept the invite and confirm that you can see the repository.

Both Student A and B: Clone the repository using your newly created SSH keys to your local environment. Make sure to clone using the link from the SSH tab. Here are the docs to help.



```
$ git clone git@github.com:dtemir/project-1-for-both.git
Cloning into 'project-1-for-both'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 2 (delta 0), pack-reused 0
Receiving objects: 100% (3/3), done.
```

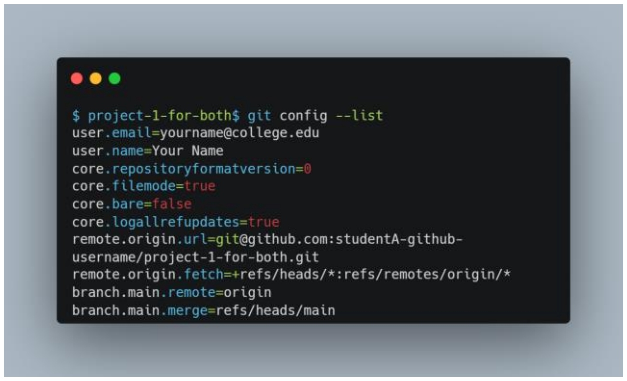
Take a screenshot of successfully cloning the repository.

Do not forget to navigate into the cloned repository and checkout the provided files using the following commands:

- **cd project-1-for-both/** and then
- **ls**

If you are looking to exceed requirements, make sure to setup the **git config** with your name and email. Use the following documentation:

- Setting your Username in git
- Setting your Commit Email Address



```
$ project-1-for-both$ git config --list
user.email=yourname@college.edu
user.name=Your Name
core.repositoryformatversion=0
core.filemode=true
core.bare=false
core.logallrefupdates=true
remote.origin.url=git@github.com:studentA-github-
username/project-1-for-both.git
remote.origin.fetch=+refs/heads/*:refs/remotes/origin/*
branch.main.remote=origin
branch.main.merge=refs/heads/main
```

Deliverables for Task 0 - Setup your local environment

Both Student A and B each are expected to deliver all of the following:

- Individually screenshots of:
  - Successfully installed git (**git --version**)
  - Successfully setup SSH auth to communicate to GitHub (**ssh -T git@github.com**)
  - Successfully cloned repository (**git clone**)
  - Exceeds requirements: successful config setup (**git config --list)**
- Links to their GitHub account
- Two sentences in responses to each question below:
  1. What is GitHub and why did you have to create a repository there?
  2. Why did you have to set up SSH and connect the public key to your account?

Task 1 - Initial Commit and Push, 2 Steps

Now that you cloned the repository, open the code and familiarize yourself with its functionality. You will edit documentation and resolve bugs. You will then update the version of the code and share it with your partner.

Committing Changes: A commit is a snapshot of your work at a particular point in time. You can always revert to this state if needed. To commit changes, use **git add** to stage changes and git commit to commit them with a message.

Pushing to Repository: After committing your changes, you need to push them to the remote repository on GitHub so that they are available to others and are stored remotely.

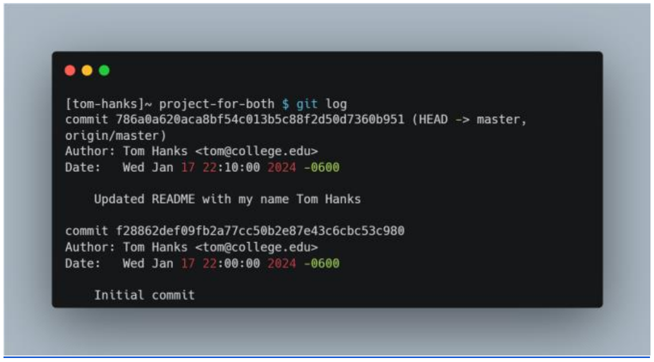
Step 1 - Small change

Start by making a small change in the repository.

Student A:

1. Edit the **README.md** file by putting your name at the end of the file.
2. Commit the change using **git add README.md** to stage the changes and then commit with **git commit -m "Updated README with my name Student A"**.
3. Push the change to GitHub using **git push** to share the version with Student B

Confirm that you made the change.

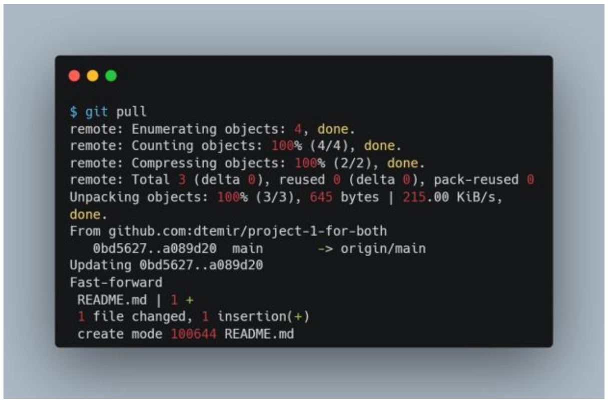


```
[tom-hanks]~ project-for-both $ git log
commit 786a0a620aca8bf54c013b5c88f2d50d7360b951 (HEAD -> master,
origin/master)
Author: Tom Hanks <tom@college.edu>
Date:   Wed Jan 17 22:10:00 2024 -0600

    Updated README with my name Tom Hanks

commit f28862def09fb2a77cc50b2e87e43c6cbc53c980
Author: Tom Hanks <tom@college.edu>
Date:   Wed Jan 17 22:00:00 2024 -0600

    Initial commit
```

*Take a screenshot of the terminal showing **git log***
Provide a link to the GitHub commit history for your repository showing the commit present

Student B:

Before making any changes, pull Student A's changes with **git pull**

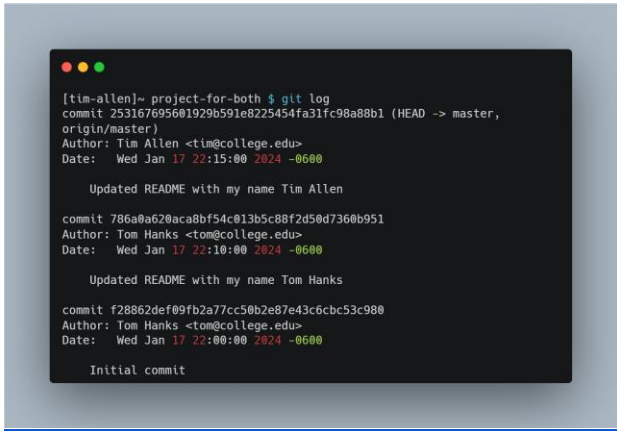


```
$ git pull
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 645 bytes | 215.00 KiB/s,
done.
From github.com:dtemir/project-1-for-both
   0bd5627..a089d20  main       -> origin/main
Updating 0bd5627..a089d20
Fast-forward
 README.md | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

Confirm that Student A's changes are in your local repository (your **git log** should look exactly the same as Student A's before you make any changes).

And make your own changes by modifying **README.md** with your name:

1. Edit the **README.md** file by putting your name at the end of the file.
2. Commit the change using **git add README.md** to stage the changes and then commit with **git commit -m "Updated README with my name Student B"**.
3. Push the change to GitHub using **git push** to share the version with Student A.



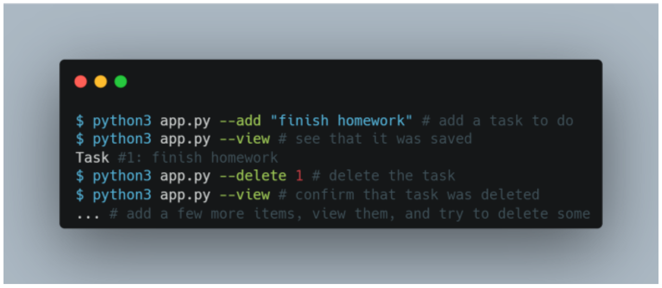
```
[tim-allen]~ project-for-both $ git log
commit 253167695601929b591e8225454fa31fc98a88b1 (HEAD -> master,
origin/master)
Author: Tim Allen <tim@college.edu>
Date:   Wed Jan 17 22:15:00 2024 -0600

    Updated README with my name Tim Allen

commit 786a0a620aca8bf54c013b5c88f2d50d7360b951
Author: Tom Hanks <tom@college.edu>
Date:   Wed Jan 17 22:10:00 2024 -0600

    Updated README with my name Tom Hanks

commit f28862def09fb2a77cc50b2e87e43c6cbc53c980
Author: Tom Hanks <tom@college.edu>
Date:   Wed Jan 17 22:00:00 2024 -0600

    Initial commit
```

*Take a screenshot of the terminal showing **git log***
Provide a link to the GitHub commit history for your repository showing the commit present

Step 2 - Bug fixes

There are two bugs present in the code. Each student needs to find one bug, commit the solution, and push it to the remote repository.

You can find the bugs by running the application a couple of times:



```
$ python3 app.py --add "finish homework" # add a task to do
$ python3 app.py --view # see that it was saved
Task #1: finish homework
$ python3 app.py --delete 1 # delete the task
$ python3 app.py --view # confirm that task was deleted
... # add a few more items, view them, and try to delete some
```

*Take a screenshot of the terminal showing **git log** with commits that solve those bugs*
Provide a link to the GitHub commit history for your repository showing the commits present

Deliverables for Task 1 - Initial Commit and Push

Both Student A and B are expected to deliver each of the following separately:

- Screenshots of:
    1. The local output of the command **git log** showing the commits made
    2. A GitHub remote repository history showing the commit pushed to remote repository
    3. Screenshot of the bug and your solution
- Two sentences on why you had to do **git pull** before you were able to push if your teammate made their changes first.

Task 2 - Branching and Pull Requests, 1 Step

Warning: as to not jump ahead, please make sure to complete this task sequentially. Start with Student A and then repeat as Student B.

Branches are used to develop features isolated from each other. The main branch is the default branch where the source code of HEAD always reflects a production-ready state.

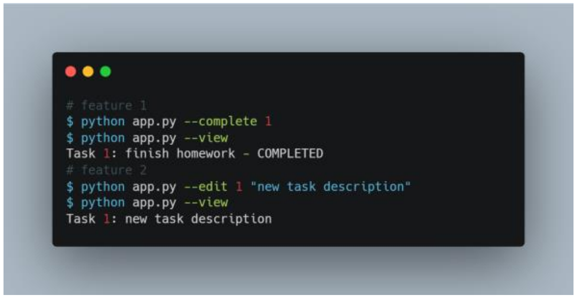
Pull Requests are used to tell others about changes you've pushed to a branch in a repository on GitHub. Once a pull request is opened, you can discuss and review the potential changes with collaborators.

Step 1 - New functionality

First, create a branch **git branch new-feature-student-A** and switch to it using **git checkout new-feature-student-A**. You may change the name of the branch based on your role.

Then, each student needs to make some changes to the code. Try adding new functionality. Some ideas:

- Add a new command, such as **python app.py --complete 1**, where the CLI will mark the task 1 as completed
- Add a new command, such as **python app.py --edit 1 "new task description"** that will let you edit the description of the task
- Feel free to come up with anything else



```
# feature 1
$ python app.py --complete 1
$ python app.py --view
Task 1: finish homework - COMPLETED
# feature 2
$ python app.py --edit 1 "new task description"
$ python app.py --view
Task 1: new task description
```

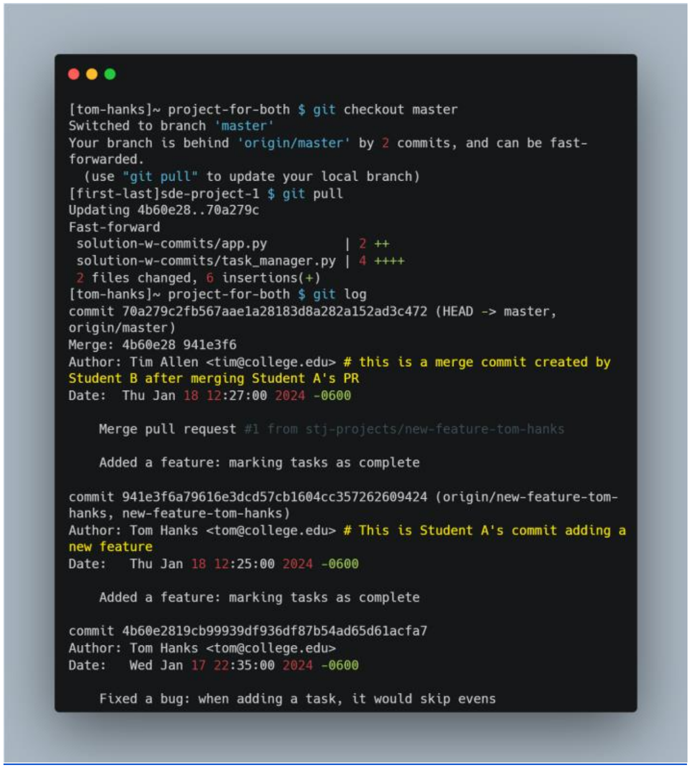
When you finish a task, commit the changes in your branch.

Push branch and create pull request, **git push origin new-feature-student-A**. Then, by going to GitHub, the repository will suggest you create a pull request. Create a pull request and fill out the name and description.

Once you create the pull request, have your partner approve and merge the pull request:

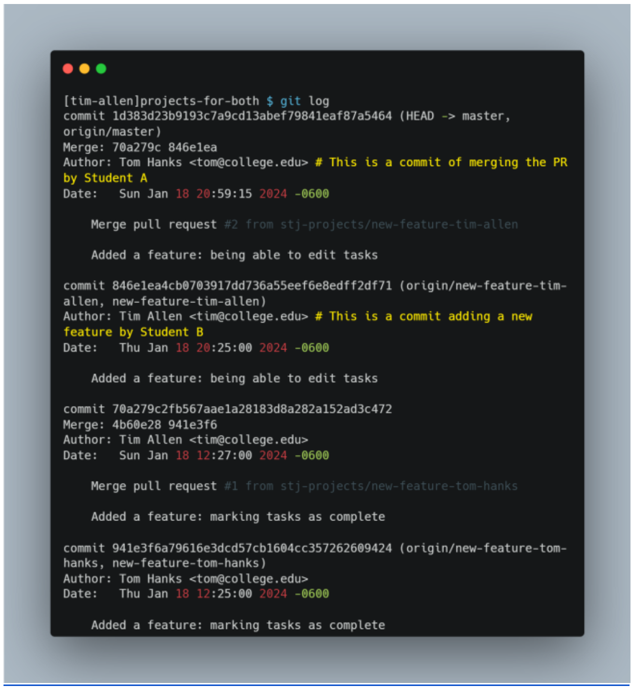
- Approving a Pull Request with Required Reviews
- Merging a Pull Request

Once merged, you can go back to the main branch and pull the changes.



```
[tom-hanks]~ project-for-both $ git checkout master
Switched to branch 'master'
Your branch is behind 'origin/master' by 2 commits, and can be fast-
forwarded.
  (use "git pull" to update your local branch)
[first-last]sde-project-1 $ git pull
Updating 4b60e28..70a279c
Fast-forward
 solution-w-commits/app.py          | 2 ++
 solution-w-commits/task_manager.py | 4 ++++
 2 files changed, 6 insertions(+)
[tom-hanks]~ project-for-both $ git log
commit 70a279c2fb567aae1a28183d8a282a152ad3c472 (HEAD -> master,
origin/master)
Merge: 4b60e28 941e3f6
Author: Tim Allen <tim@college.edu> # this is a merge commit created by
Student B after merging Student A's PR
Date:  Thu Jan 18 12:27:00 2024 -0600

    Merge pull request #1 from stj-projects/new-feature-tom-hanks

    Added a feature: marking tasks as complete

commit 941e3f6a79616e3dcd57cb1604cc357262609424 (origin/new-feature-tom-
hanks, new-feature-tom-hanks)
Author: Tom Hanks <tom@college.edu> # This is Student A's commit adding a
new feature
Date:   Thu Jan 18 12:25:00 2024 -0600

    Added a feature: marking tasks as complete

commit 4b60e2819cb99939df936df87b54ad65d61acfa7
Author: Tom Hanks <tom@college.edu>
Date:   Wed Jan 17 22:35:00 2024 -0600

    Fixed a bug: when adding a task, it would skip evens
```

This is what it should look like after Student B added a new feature:



```
[tim-allen]projects-for-both $ git log
commit 1d383d23b9193c7a9cd13abef79841eaf87a5464 (HEAD -> master,
origin/master)
Merge: 70a279c 846e1ea
Author: Tom Hanks <tom@college.edu> # This is a commit of merging the PR
by Student A
Date:   Sun Jan 18 20:59:15 2024 -0600

    Merge pull request #2 from stj-projects/new-feature-tim-allen

    Added a feature: being able to edit tasks

commit 846e1ea4cb0703917dd736a55eef6e8edff2df71 (origin/new-feature-tim-
allen, new-feature-tim-allen)
Author: Tim Allen <tim@college.edu> # This is a commit adding a new
feature by Student B
Date:   Thu Jan 18 20:25:00 2024 -0600

    Added a feature: being able to edit tasks

commit 70a279c2fb567aae1a28183d8a282a152ad3c472
Merge: 4b60e28 941e3f6
Author: Tim Allen <tim@college.edu>
Date:   Sun Jan 18 12:27:00 2024 -0600

    Merge pull request #1 from stj-projects/new-feature-tom-hanks

    Added a feature: marking tasks as complete

commit 941e3f6a79616e3dcd57cb1604cc357262609424 (origin/new-feature-tom-
hanks, new-feature-tom-hanks)
Author: Tom Hanks <tom@college.edu>
Date:   Thu Jan 18 12:25:00 2024 -0600

    Added a feature: marking tasks as complete
```

Deliverables for Task 2 - Branching and Pull Requests

- Screenshots of:

  1. Creating a local branch for a specific bug fix (**git branch -a**)
  2. Pushing a commit with fixes to the branch (**git log**)
  3. Exceeds requirements: setup a branch protection rule to stop anyone from pushing commits to main. Reference: and provide a screenshot (one screenshot should be enough for both students)

- Link to the Pull Requests raised on GitHub and **merged** into main
- Two sentences on the purpose of Pull Requests and Branches

Task 3 - Using Git Stash, 1 Step

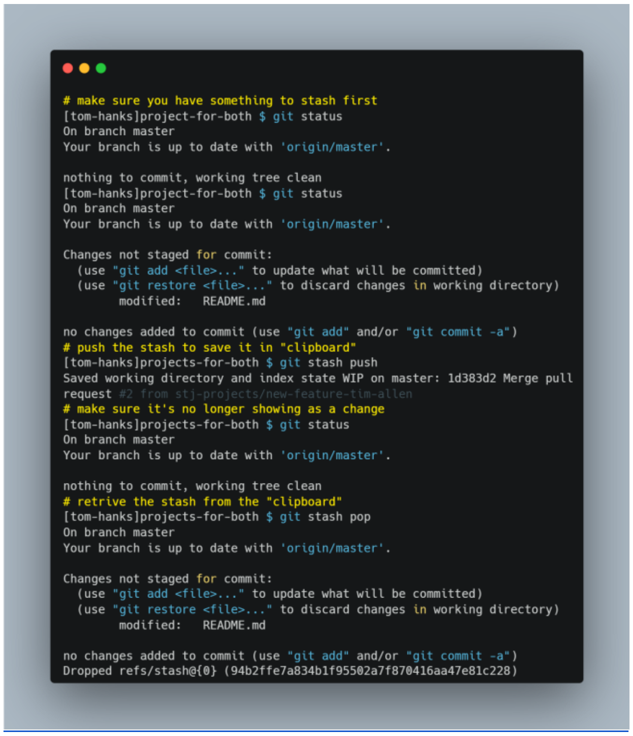
Git Stash is used for saving your modified and staged changes and reverting the working directory to a clean state. It's like a clipboard for uncommitted changes. Here are docs to help.

Step 1 - Stash and Pop

1. Make any changes in your code. Save the files but don't commit them.
2. Use **git stash push** to stash your changes.
3. Reapply the changes with **git stash pop**.

Take screenshots showing the use of git stash push and git stash pop.

Exceeds Requirements: try using functions like **git stash list** to show multiple stashes and **git stash drop** to show that you deleted stashes.



```
# make sure you have something to stash first
[tom-hanks]project-for-both $ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
[tom-hanks]project-for-both $ git status
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
# push the stash to save it in "clipboard"
[tom-hanks]projects-for-both $ git stash push
Saved working directory and index state WIP on master: 1d383d2 Merge pull
request #2 from stj-projects/new-feature-tim-allen
# make sure it's no longer showing as a change
[tom-hanks]projects-for-both $ git status
On branch master
Your branch is up to date with 'origin/master'.

nothing to commit, working tree clean
# retrive the stash from the "clipboard"
[tom-hanks]projects-for-both $ git stash pop
On branch master
Your branch is up to date with 'origin/master'.

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (94b2ffe7a834b1f95502a7f870416aa47e81c228)
```

Deliverables for Task 3 - Using Git Stash

- Screenshots of:

    1. Use of **git stash apply** to save uncommitted changes.
    2. Use of **git stash pop** from git stash to apply stashed changes back into the working directory.
    3. Exceeds Requirements: show the use of **git stash list** with multiple stashes and **git stash drop.**

- Two sentences with a brief explanation of when and why to use git stash.

Task 4 - Handling Merge Conflicts, 6 Steps

Merge conflicts occur when different branches have conflicting changes in the same part of the same file. They must be resolved manually before merging.

Step 1 - Checkout and Pull

Before you start, make sure you're on the **main** branch after you successfully merge your pull requests: **git checkout main & git pull**

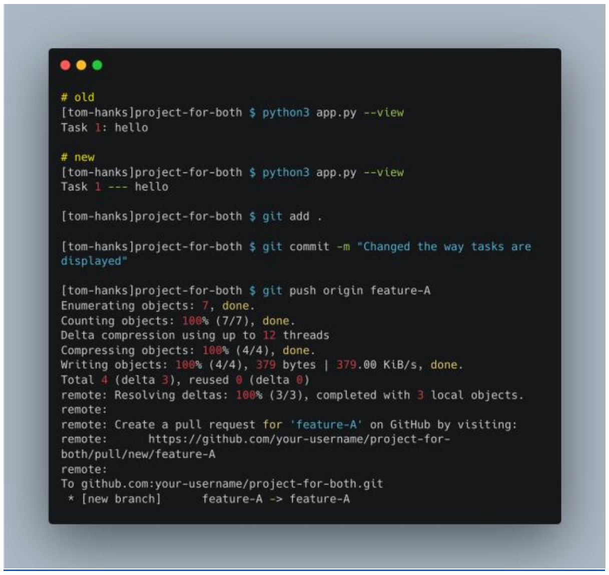
Step 2 – Each Create a New Branch

Student A and B: Create a New Branch Each:

1. Student A: **git checkout -b feature-A**
2. Student B: **git checkout -b feature-B**

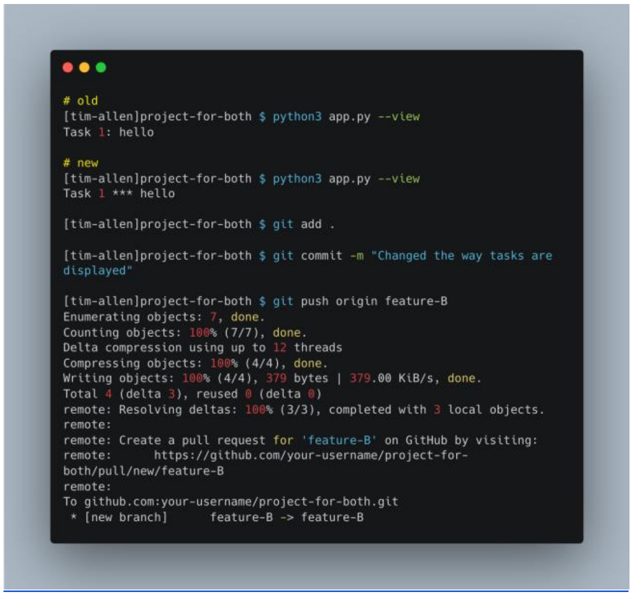
Step 3 - Make Conflicting Code Changes

Student A: Modify a specific function in **task_manager.py**. Specifically, change the format of how tasks are displayed in the **view_tasks** method. Instead of displaying them with:



```
# old
[tom-hanks]project-for-both $ python3 app.py --view
Task 1: hello

# new
[tom-hanks]project-for-both $ python3 app.py --view
Task 1 --- hello

[tom-hanks]project-for-both $ git add .

[tom-hanks]project-for-both $ git commit -m "Changed the way tasks are
displayed"

[tom-hanks]project-for-both $ git push origin feature-A
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 379 bytes | 379.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'feature-A' on GitHub by visiting:
remote:      https://github.com/your-username/project-for-
both/pull/new/feature-A
remote:
To github.com:your-username/project-for-both.git
 * [new branch]      feature-A -> feature-A
```

Student B: Modify <u>the same</u> function **Student A** did in **task_manager.py** but with a different symbol. Specifically, change it from old to new:



```
# old
[tim-allen]project-for-both $ python3 app.py --view
Task 1: hello

# new
[tim-allen]project-for-both $ python3 app.py --view
Task 1 *** hello

[tim-allen]project-for-both $ git add .

[tim-allen]project-for-both $ git commit -m "Changed the way tasks are
displayed"

[tim-allen]project-for-both $ git push origin feature-B
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 12 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (4/4), 379 bytes | 379.00 KiB/s, done.
Total 4 (delta 3), reused 0 (delta 0)
remote: Resolving deltas: 100% (3/3), completed with 3 local objects.
remote:
remote: Create a pull request for 'feature-B' on GitHub by visiting:
remote:      https://github.com/your-username/project-for-
both/pull/new/feature-B
remote:
To github.com:your-username/project-for-both.git
 * [new branch]      feature-B -> feature-B
```

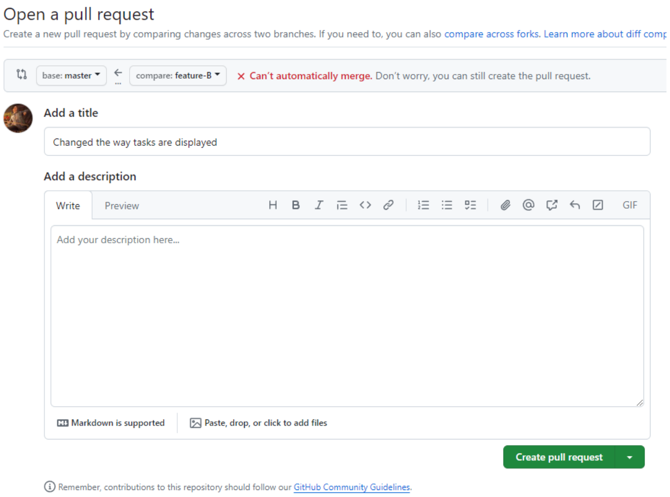
Both Students: Commit the changes to your branch and push to GitHub.

Step 4 – Attempt to Merge

Attempt to Merge Both Branches into Main:

a. **Student A**: Merge **feature-A** into Main on GitHub.

b. **Student B:** Attempt to merge **feature-B** into Main. This will result in a merge conflict because of the changes made by Student A in the same area of the code.



Open a pull request

Create a new pull request by comparing changes across two branches. If you need to, you can also compare across forks. Learn more about diff comp

base: master ← compare: feature-B ✕ Can't automatically merge. Don't worry, you can still create the pull request.

Add a title

Changed the way tasks are displayed

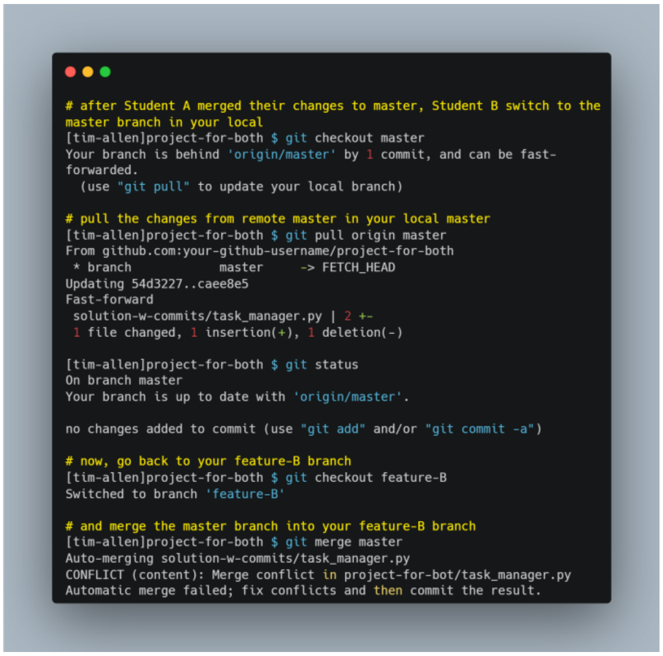
Add a description

Add your description here...

ⓘ Remember, contributions to this repository should follow our GitHub Community Guidelines.

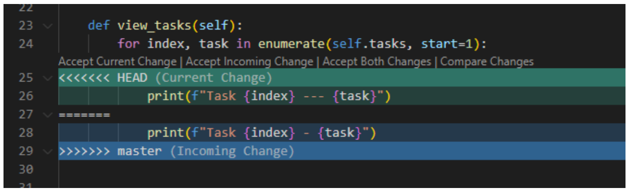
Step 5 - Resolve the Conflict

Student B: Manually resolve the conflict by editing **task_manager.py** to incorporate changes from both students. Then complete the merge. Here are docs to help.



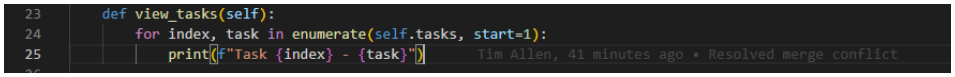
```
# after Student A merged their changes to master, Student B switch to the
master branch in your local
[tim-allen]project-for-both $ git checkout master
Your branch is behind 'origin/master' by 1 commit, and can be fast-
forwarded.
  (use "git pull" to update your local branch)

# pull the changes from remote master in your local master
[tim-allen]project-for-both $ git pull origin master
From github.com:your-github-username/project-for-both
 * branch            master     -> FETCH_HEAD
Updating 54d3227..caee8e5
Fast-forward
 solution-w-commits/task_manager.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)

[tim-allen]project-for-both $ git status
On branch master
Your branch is up to date with 'origin/master'.

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

# now, go back to your feature-B branch
[tim-allen]project-for-both $ git checkout feature-B
Switched to branch 'feature-B'

# and merge the master branch into your feature-B branch
[tim-allen]project-for-both $ git merge master
Auto-merging solution-w-commits/task_manager.py
CONFLICT (content): Merge conflict in project-for-bot/task_manager.py
Automatic merge failed; fix conflicts and then commit the result.
```

The conflict can be seen in task_manager.py:


Accept Current Change | Accept Incoming Change | Accept Both Changes | Compare Changes

```
    def view_tasks(self):
        for index, task in enumerate(self.tasks, start=1):
<<<<<<< HEAD (Current Change)
            print(f"Task {index} --- {task}")
=======
            print(f"Task {index} - {task}")
>>>>>>> master (Incoming Change)
```

Choose the incoming change (the one that follows ===== and stopped by >>>>>) and commit the change. The issue is now resolved.



```
    def view_tasks(self):
        for index, task in enumerate(self.tasks, start=1):
            print(f"Task {index} - {task}")    Tim Allen, 41 minutes ago • Resolved merge conflict
```

After resolving, commit the changes and push to GitHub. Now, you can successfully create a PR and get it merged to main.

Step 6 - Reverse Roles and Repeat

Having merged both **feature-A** and **feature-B** into main, create new branches off main called **feature-A2** and **featureB2** and make any small modification to the codebase (like the functionality to print tasks) but this time reverse roles.

Make sure that **Student B** gets their changes merged into main first, and **Student A** resolves the conflict in their branch before merging it into main.

<u>Deliverables for Task 4 – Handling Merge Conflicts</u>

- Screenshots of:
  1. Intentionally introducing merge conflicts for each other (by following instructions and using branches beyond main)
  2. Resolving the merge conflict locally

- Links to merge commits.
- Four sentences on why merge conflicts happen, how to resolve them, and the strategy you used.

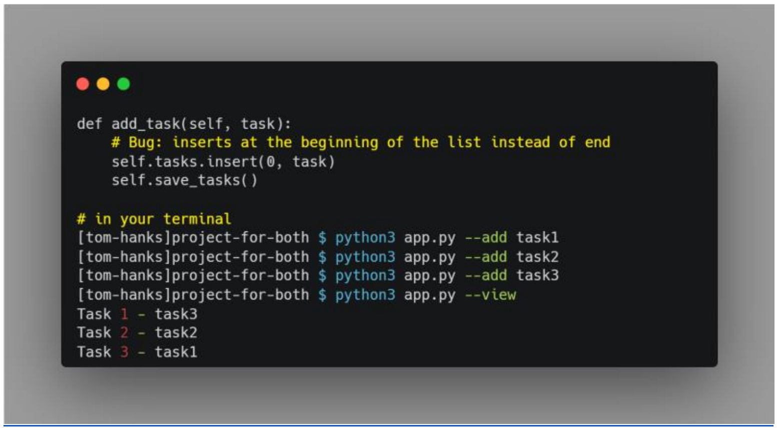
<u>Task 5 - Revert Changes, 2 Steps</u>

Reverting is used to undo changes by creating a new commit that reverses changes made in a previous commit.

Step 1 – Introducing a Bug

Student A and B: Make a deliberate error in the code, such as changing a function in **task_manager.py** to behave incorrectly. Create the commit in your local repository.

Specifically, **Student A**, change the **add_task** function to insert element at the start of the list instead of the end



```
def add_task(self, task):
    # Bug: inserts at the beginning of the list instead of end
    self.tasks.insert(0, task)
    self.save_tasks()

# in your terminal
[tom-hanks]project-for-both $ python3 app.py --add task1
[tom-hanks]project-for-both $ python3 app.py --add task2
[tom-hanks]project-for-both $ python3 app.py --add task3
[tom-hanks]project-for-both $ python3 app.py --view
Task 1 - task3
Task 2 - task2
Task 3 - task1
```

Student B: change the **delete_task** function to delete elements at the start of the list

Step 2 – Revert the Commit

Revert the commit. Use **git revert <commit-hash>** to undo this change, creating a new commit. You can find the commit hash by using **git log**. It is a long number you see such as **60cb1c16f5fd8f9dde13164626ba0b89cb56f37c**.

Deliverables for Task 5 - Revert Changes

- Screenshot of:
  - Running git revert and showing the commit is absent in local
- Link to the pull request and commit those reverted changes on GitHub (their commit)
- Two sentences on why reverting is useful. Consider its use in companies that deploy software products daily and may make a mistake once in a while.

Task 6 - Creating and Using Tags, 2 Steps

Tags are used to mark specific points in a repository's history, typically used for version releases. G

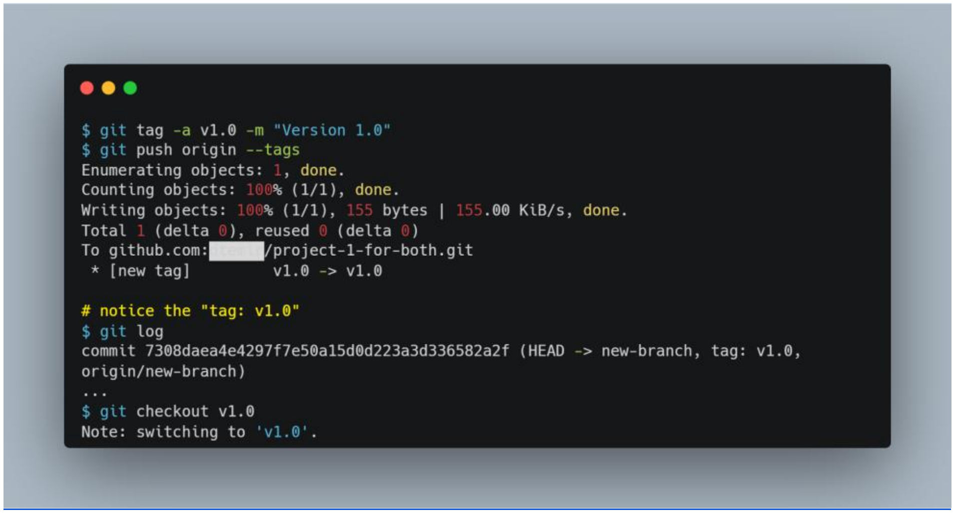
Step 1 – Create a Tag

Student A and B: Create a tag. Having completed all of the above tasks and making sure that everything works, create a tag. Use **git tag -a v1.0 -m "Version 1.0 by Student A/B"** for an annotated tag or simply **git tag v1.0** for a lightweight tag.

Step 2 – Push to GitHub

Push the tags to GitHub. Use **git push origin --tags** to push the tags to the remote repository

You can confirm you created a tag by checking it out. Use **git checkout v1.0** to view the state of the repository at that tag.



```
$ git tag -a v1.0 -m "Version 1.0"
$ git push origin --tags
Enumerating objects: 1, done.
Counting objects: 100% (1/1), done.
Writing objects: 100% (1/1), 155 bytes | 155.00 KiB/s, done.
Total 1 (delta 0), reused 0 (delta 0)
To github.com:[redacted]/project-1-for-both.git
 * [new tag]         v1.0 -> v1.0

# notice the "tag: v1.0"
$ git log
commit 7308daea4e4297f7e50a15d0d223a3d336582a2f (HEAD -> new-branch, tag: v1.0,
origin/new-branch)
...
$ git checkout v1.0
Note: switching to 'v1.0'.
```

Once you push the tags to the remote repository, you will be able to find them in your GitHub repository.

Take a screenshot of creating the tags in your terminal

Deliverables for Task 6 - Creating and Using Tags

- Screenshots of:
  - Creation of annotated and/or lightweight tags in your local Git repository.
  - Push the tag to the remote GitHub repository.
  - Checking out the specific tag locally.

- Two sentences with a brief explanation of tags and the difference between annotated and lightweight tags.