Q&A for Bash and Git

Question

Why Rstudio did not pick updated version of R?

RStudio does not pick up the updated version of R automatically, you need to do it manually. In RStudio, Go to Tools then `Global Options`, then click `Change` under `R version` and from the drop down menu select the one version you need.

Question

Do we have GitHub classroom?

We do not have a GitHub classroom. Please set up your own GitHub.

What is the difference between these two?

GitHub is the general-purpose platform for version control and collaboration in software development, while GitHub Classroom is a specialized tool for managing coding assignments in an educational setting.

We may choose to use it later depends on department requirement.

Question

Does anybody know how to establish a working directory? I had tried to run "cat file1.txt" and the output said "No such file or directory".

1. Double-check your command for any typos.
2. Note that the working directory in Terminal/Bash and RStudio Terminal pane is different.

3. If using RStudio Terminal, ensure the file is in the RStudio Terminal's current directory.
4. Use `ls` to see if `file1.txt` is in the current directory.
5. If it's there, `cat file1.txt` should work; if not, use `pwd` to identify your current directory.
6. Download `file1.txt` to that directory and confirm with `ls`.
7. Finally, execute `cat file1.txt` once you've confirmed the file's presence.

Question

How to install text editor

I recommend installing Visual Studio or Notepad++ or Sublime as your editor on your laptop

- **Visual Studio:**

Install Visual Studio Code (VS Code). For more information, see "Setting up VS Code" in the VS Code documentation.

Open Git Bash/Terminal Type:

```
git config --global core.editor "code --wait"
```

- **Notepad++**

Install Notepad++ from https://notepad-plus-plus.org/. For more information, see "Getting started" in the Notepad++ documentation.

Open Git Bash/Terminal Type:

```
git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'
 -multiInst -notabbar -nosession -noPlugin"
```

- **Sublime**

Install Sublime Text. For more information, see"Installation" in the Sublime Text documentation.

Open Git Bash/Terminal

Type:

```
git config --global core.editor "'C:/Program Files (x86)/sublime text 3/subl.exe' -w"
```

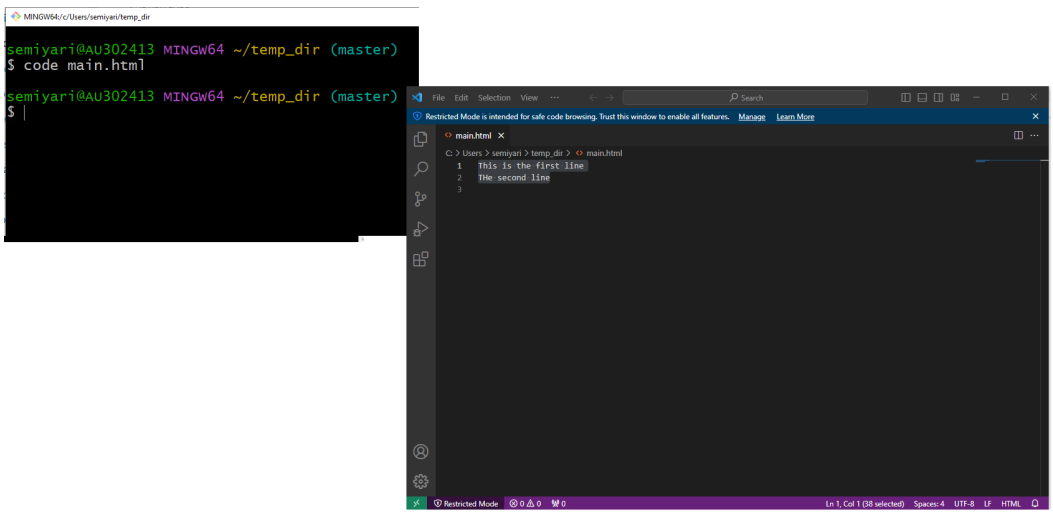
Please note that the command `code` will open your text editor. For instance, if you have a file called `main.html` and inside of this file you have

```
This is the first line
THe second line
```

if you type

```
code main.html
```

Then file `main.html` will be opened by your text editor (if you have such a file in your directory)



Now we want to create the .gitignore file:

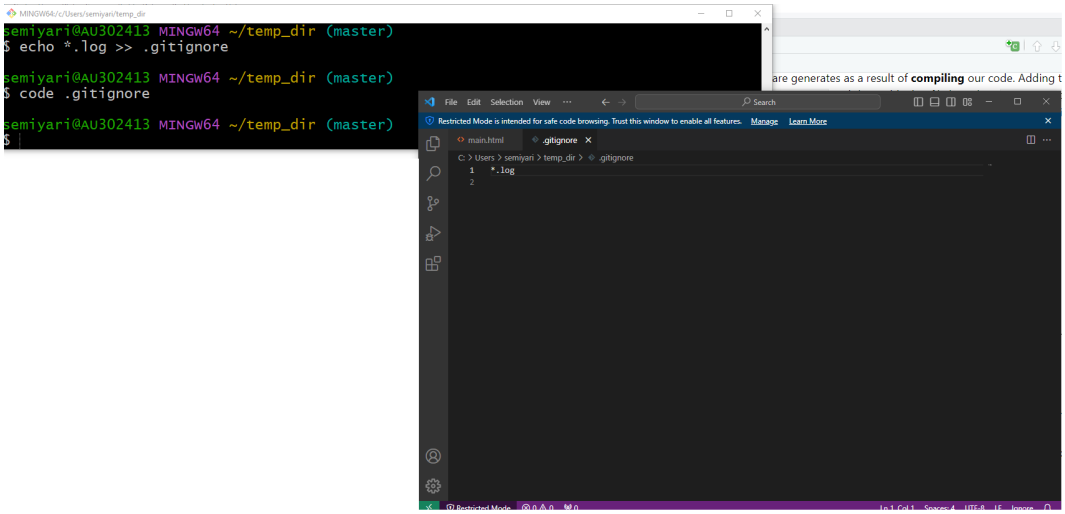
1. You need to create the file. Type

```
echo *.log >> .gitignore
```

This will write `*.log` inside of the `.gitignore` and Git will ignore this file. Then

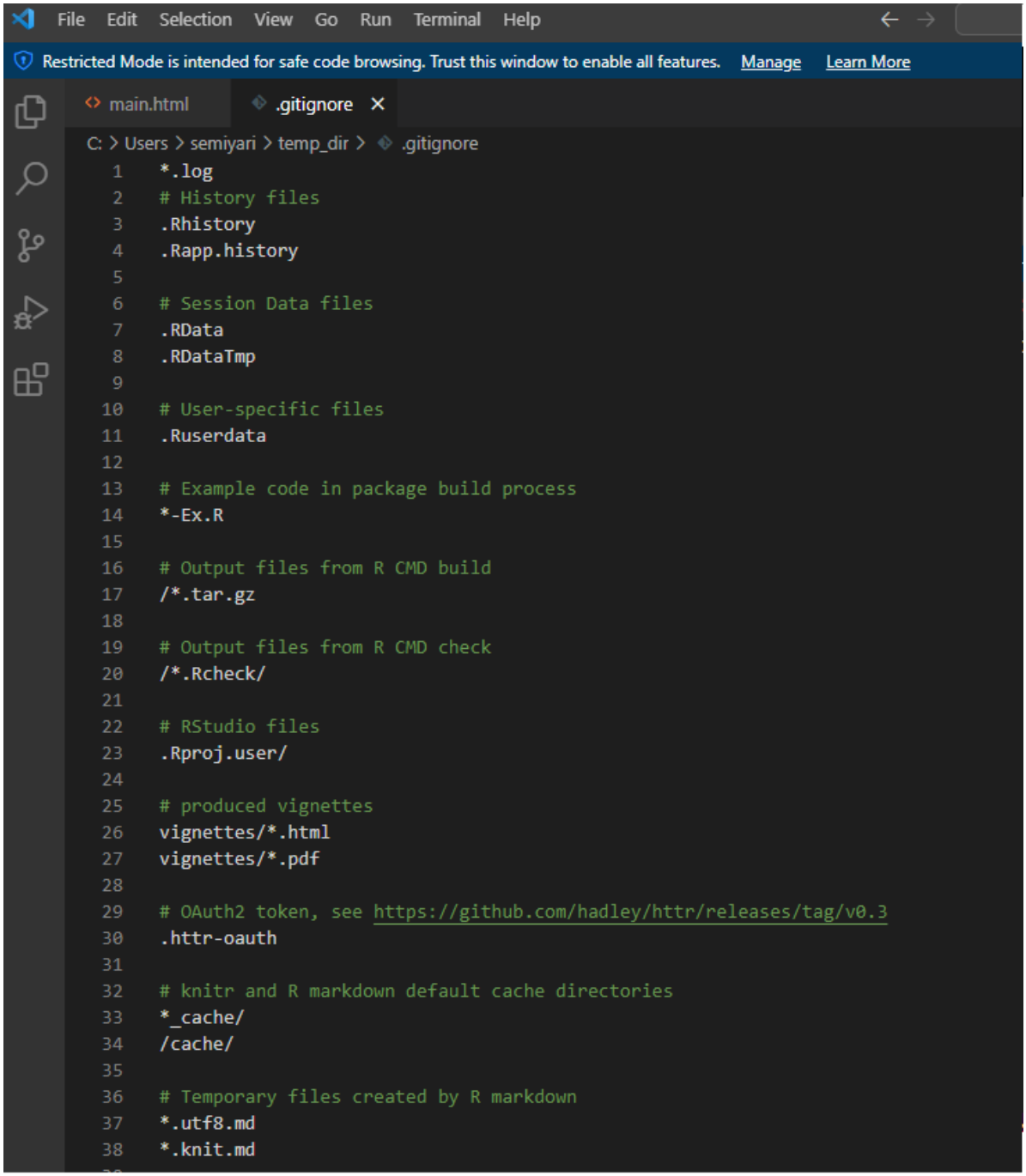
```
code .gitignore
```

will open `.gitignore` by your text editor. You will see inside of your text editor is `*.log`



If you go to github.com/github/gitignore you can see various "gitignore" template for different programming lnguage. For R is the file "R.gitignore".

I have added everything in "R.gitignore" into my `.gitignore` and saved the file.



Then

- Type `git add .gitignore` and
- `git commit -m "Adding gitignore"`.

- So this is how git ignores a file or directory. But remember, it works if you have not already included a file or directory in you repository. If you have files added before creating `.gitignore`, those file will stay in your repository.

Question

** What to do after installing the test edittor?**

After you have installed your text editor. I would like you to open the .gitconfig file. Where is its location? .gitconfig file found under the user's home directory.

Where is your home directory?

Windows User: Open Bash and type

```
pwd
```

Mac User: Open Terminal and type

```
pwd
```

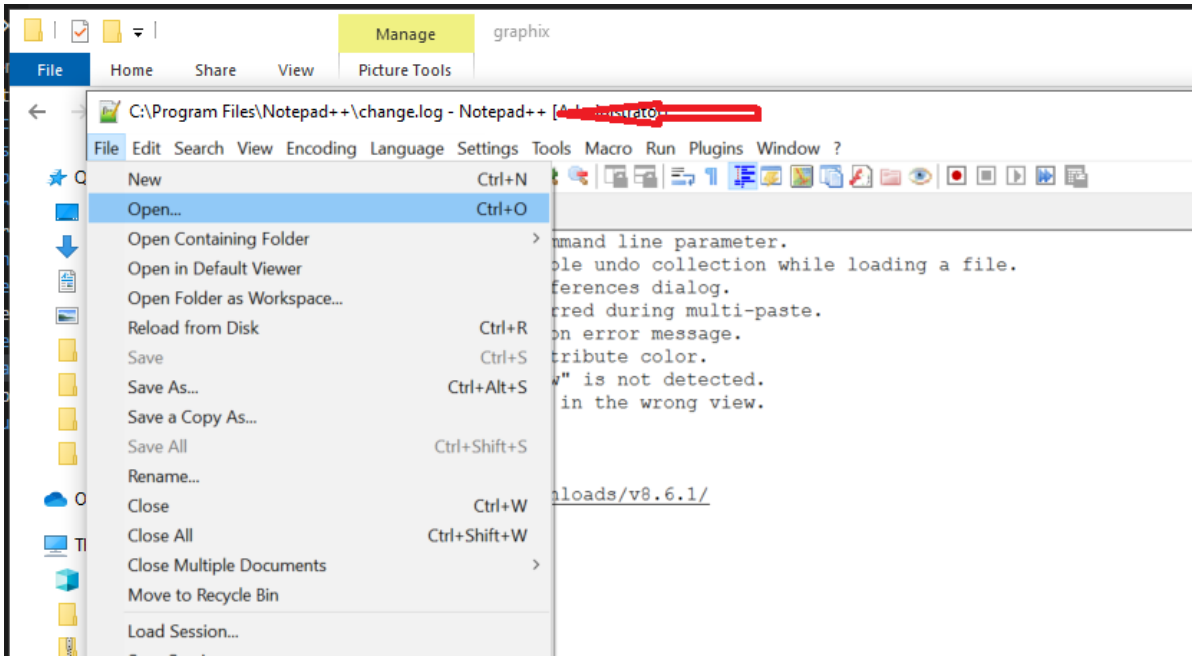
The result is your home directory. (The home directory and working directory are different).

How to open `.gitconfig`file?

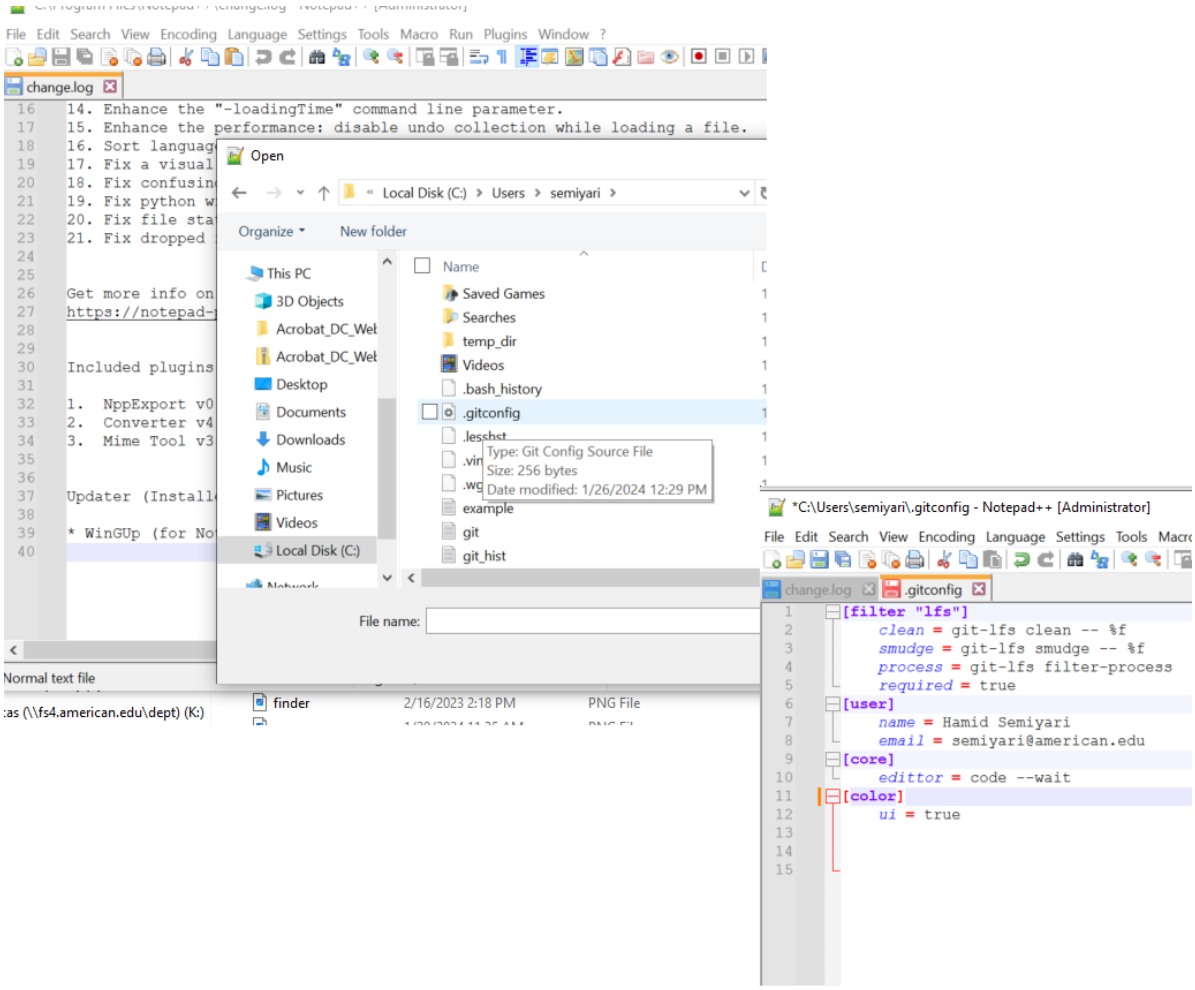
- you can go to your home directory find `.gitconfig` and open it.

Or

- You can open your editor (in this case I am going to open the Notepad++ editor, to show you the choice of editor is not very important and it is merely a matter of preference) and then in your editor click "open" and from there go to your home directory and open `.gitconfig`



Yours should be similar to

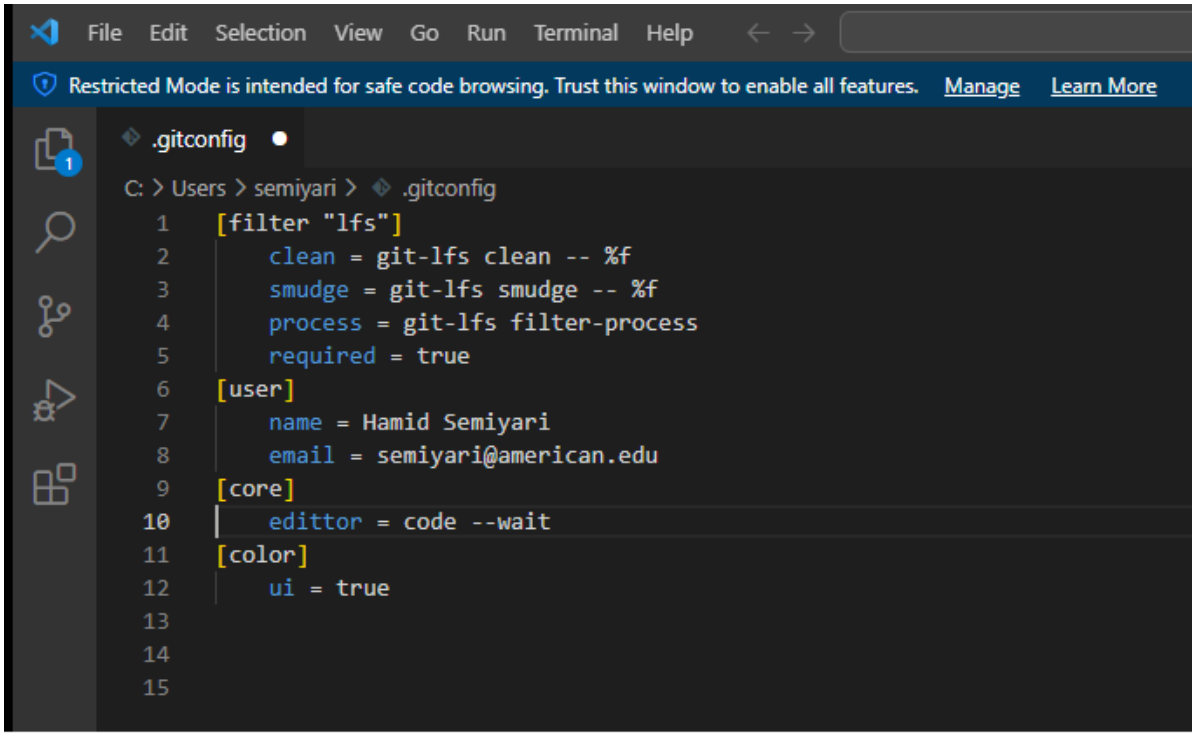


- Or you may type

```
code .gitconfig
```

To open the `.gitconfig` file.

Since I use Visual Studio as a default editor, thus from the command line code `.gitconfig` I will get



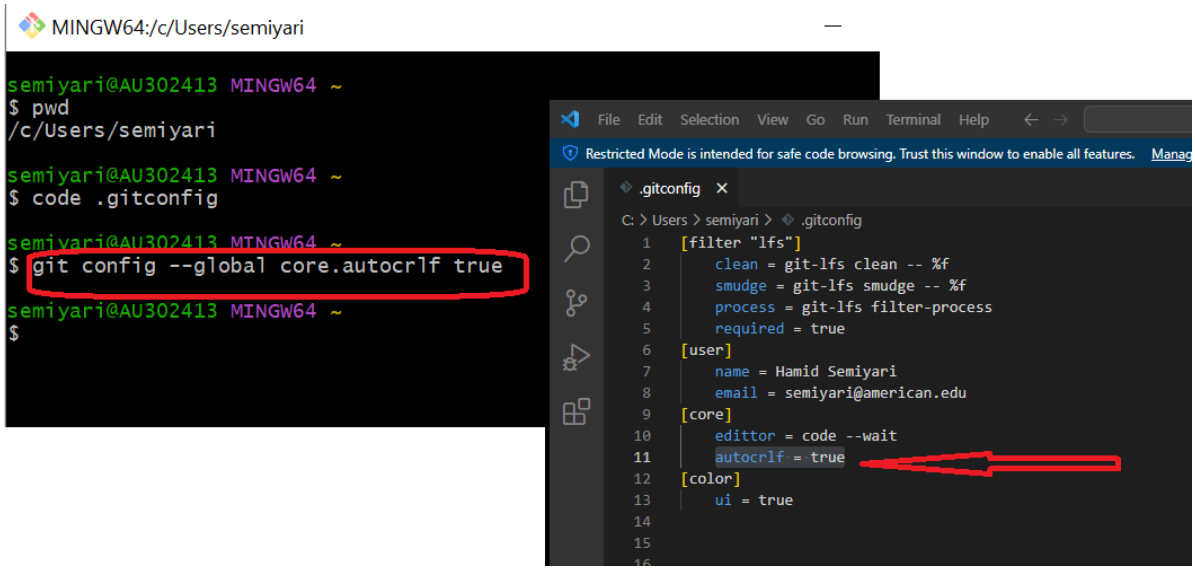
Important note

- **Handle end of lines:** This is a very important setting that lots of people miss.
  - On Windows end of lines are marked with two special characters `\r` and `\n`.
  - Carriage Return `\r`: It means that the cursor should go back to the beginning of the line.
  - Line feed `\n`: It means that the curser must go to the next line
  - On Mac and Linux systems, the end of lines are indicated with - Line feed `\n`

That means if we don't handle end of lines properly we are going to run into some issues down the road.

- **Window:** `git config --global`core.autocrlf true (crlf is short for Carriage Return Line Fit)
- *Mac or Linux:** `git config --global` core.autocrlf input

Since I have Windows I have typed `git config --global core.autocrlf true` and as you can see this line "autocrlf = true" has been added to my `.gitconfig` file



Question

Does this approach work to copy files? `output.txt >> output2.txt`

The operand `>>` means "append" (add as an attachment or supplement).

Example:

'echo TEST » file1.txt

If you type

`cat file1.txt` it returns the content of the file1.txt which is only the word "TEST"

The command `file1.txt >> file2.txt` creates a file but it is a blank file. What I mean is if you type on your Terminal `cat file2.txt` you get nothing.

Copy a file:

Use `cp file1.txt file2.txt` to create a copy of `file1.txt` which is called `file2.txt`.

Move/Rename a file

Use `mv file1.txt file2.txt` This command will remove the `file1.txt` and create a new file called `file2.txt`. Or we can say the command rename the `file1.txt` to `file2.txt`

Question

What is difference between >> and > in Bash/Terminal?

- `>`

This symbol `>` overwrites the file if it exists or creates it if it doesn't exist.

```
echo "The \">\" overwrites the file if it exists or creates it if it doesn't exist." > file1
```

Run

```
cat file1.txt
```

It will return

The ">" overwrites the file if it exists or creates it if it doesn't exist.

If you run this code

```
echo "This will overwrite the file and what we had before has been replaced with this line" >
```

Type on your terminal

```
cat file1.txt
```

Will return

This will overwrite the file and what we had before has been replaced with this line

- `>>` This `>>` is used to append to a file. Run the following

echo "This symbol "»" is used to append to a file." » file1.txt

if you run `cat file1.txt` you will see both lines are in file1.txt

```
cat file1.txt
```

it returns: This will overwrite the file and what we had before has been replaced with this line

This symbol "»" is used to append to a file.

Moreover

The `>>` operator in Bash is employed to append the output of a command or text to a file. It is commonly used to append content, such as a sentence or the output of a command, to the end of a file. For instance, if you wish to add the entire content of `file1.txt` to `file2.txt`, you can use the `cat` command along with the `>>` operator:

```
cat file1.txt >> file2.txt
```

Let's verify this:

```
echo "Hi" >> file1.txt
```

Executing `cat file1.txt` will return "Hi," but attempting `file1.txt >> file2.txt` will not append the content of file1.txt to file2.txt. In fact, it will result in an error in the Bash terminal, and in the RStudio terminal, you will end up with only a blank file. The correct approach is to use:

```
cat file1.txt >> file2.txt
```

This will add the content of file1.txt to file2.txt, and executing `cat file2.txt` will return "Hi."

Let's take it a step further:

```
echo "This is the second line" >> file1.txt
```

Now, executing `cat file1.txt` will return:

```
Hi
This is the second line
```

And executing `cat file1.txt >> file2.txt` will add these two lines to file2.txt. Thus, `cat file2.txt` will return:

```
Hi
Hi
This is the second line
```

If you use the > operator instead, like this:

```
cat file1.txt > file2.txt
```

It will override the content of `file2.txt` with that of `file1.txt`, effectively giving you a copy of `file1.txt`. Thus, `cat file2.txt` will return:

```
Hi
This is the second line
```

Question

I am familiar with GitHub, Do I need to learn Git commands?

- GitHub is an online platform designed for hosting and collaborating on Git repositories.
- Git, the underlying version control system, powers GitHub's operations.
- If you've used GitHub to collaborate on projects and review code, you might be less familiar with Git commands for managing local repositories, branching, merging, and other tasks. Understanding these Git commands and workflows is crucial for a deeper grasp of software development, particularly if you're already using GitHub extensively.
- I strongly recommend learning Git for anyone serious about advancing their skills in software development.

Question

What are git stashing changes?

To show stashing, let's go through a practical example step-by-step:

1. Create a Repository and Initial Commit
2. Modify the File and Check Status
3. You will see that README.md is modified.

```
semiyari@AU302413 MINGW64 ~/Test (master)
$ git init Test
Initialized empty Git repository in C:/Users/semiyari/Test/Test/.git/

semiyari@AU302413 MINGW64 ~/Test (master)
$ cd Test

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ ls

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ ls -a
./  ../  .git/

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ touch README

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch master

No commits yet

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

nothing added to commit but untracked files present (use "git add" to track)

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git add README

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git commit -m "Initial"
[master (root-commit) c935527] Initial
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ echo "This is practice for Stashing" > README

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch 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

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

4. Imagine your boss asks you to work on something else urgently, but you don't want to commit the incomplete changes in README.md.

5. You stash the file and a snapshot of the file will be stored locally (whereas the commit is part of the public git history).

`git stash` command saves your changes and reverts your working directory to the last commit.

6. Create a new file (test1) and commit it

7. Check the log to see the history of all your commits.

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

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git stash
warning: in the working copy of 'README', LF will be replaced by CRLF the next t
ime Git touches it
Saved working directory and index state WIP on master: c935527 Initial

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch master
nothing to commit, working tree clean

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ echo "This file is new" >> test1

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch master
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        test1

nothing added to commit but untracked files present (use "git add" to track)

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git add .
warning: in the working copy of 'test1', LF will be replaced by CRLF the next ti
me Git touches it

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git commit -m "I Add and Commit test1 after stashing README"
[master 9f1c81c] I Add and Commit test1 after stashing README
 1 file changed, 1 insertion(+)
 create mode 100644 test1

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git log
commit 9f1c81cd44fd4f390968ea48414151e94f5f8d57 (HEAD -> master)
Author: Hamid Semiyari <semiyari@american.edu>
Date:   Sun Jul 7 16:21:45 2024 -0400

    I Add and Commit test1 after stashing README

commit c93552723c4ffcce125cb22af5e2c2e1e488ef28
Author: Hamid Semiyari <semiyari@american.edu>
Date:   Sun Jul 7 16:12:35 2024 -0400

    Initial

semiyari@AU302413 MINGW64 ~/Test/Test (master)
```

8. Type `git status` You will see "nothing to commit, working tree clean".

9. `git stash pop` command restores the README modifications. If there were no conflicts, README will appear modified again.

10. `git status` You will see that README is modified.

```
semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch master
nothing to commit, working tree clean

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git stash pop
On branch 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

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (302191865e07430a81ae5bb6ad87a6850883c2a1)

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$ git status
On branch 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

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

semiyari@AU302413 MINGW64 ~/Test/Test (master)
$
```

https://youtu.be/KDKLNvKQ4MU

Question

Is it ever useful or okay to squash multiple commits down into one?

- Commit frequently and keep commits focused on individual issues. This practice enhances clarity, reversibility, collaboration, and history tracking by ensuring each commit represents a single logical change.

Question

How to create a repository in my laptop and push it to my GitHub?

There are two main scenarios for GitHub operations:

1. Cloning from GitHub to Local Machine:

- If you have a repository on GitHub and want to work with it locally, you clone it to your machine.

2. Creating and Pushing from Local Machine to GitHub:

- If you have a folder on your local machine that you want to store as a new repository on GitHub, you push it there for collaboration or safekeeping.

We have already created a repository on GitHub and cloned it to our local machine.

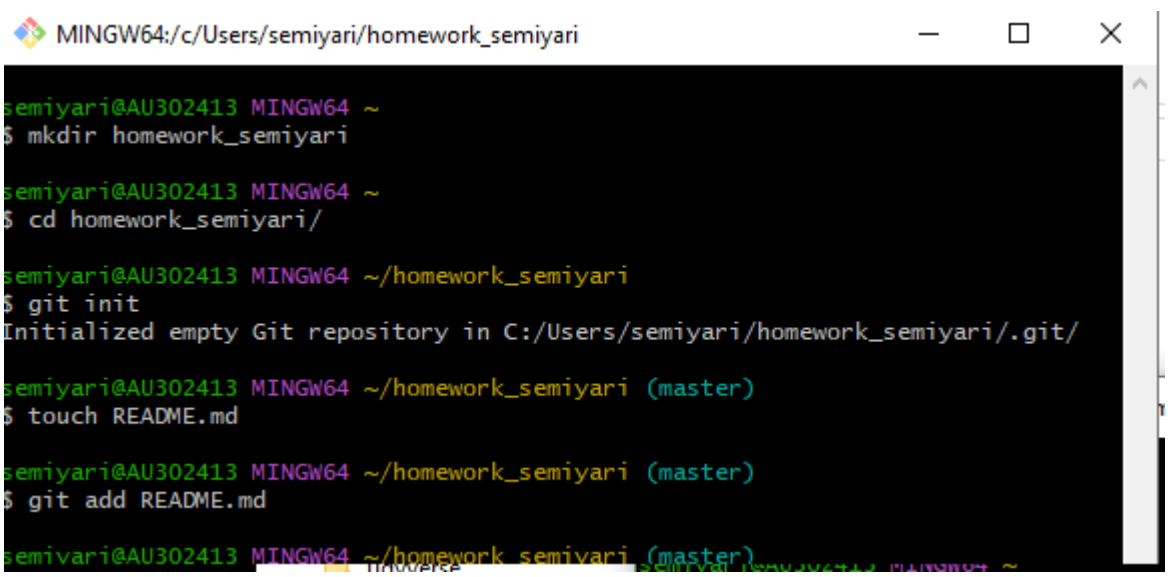
Now, we'll reverse the process: we'll create a repository on our local machine and then push it to GitHub. All operations should be performed using terminal/bash. Follow these steps to begin:

I. Create a Local Repository:

Start by creating a directory named `homework_semiyari` (replace `semiyari` with your last name). Navigate into this directory: `cd homework_semiyari`. Initialize a Git repository: `git init`.

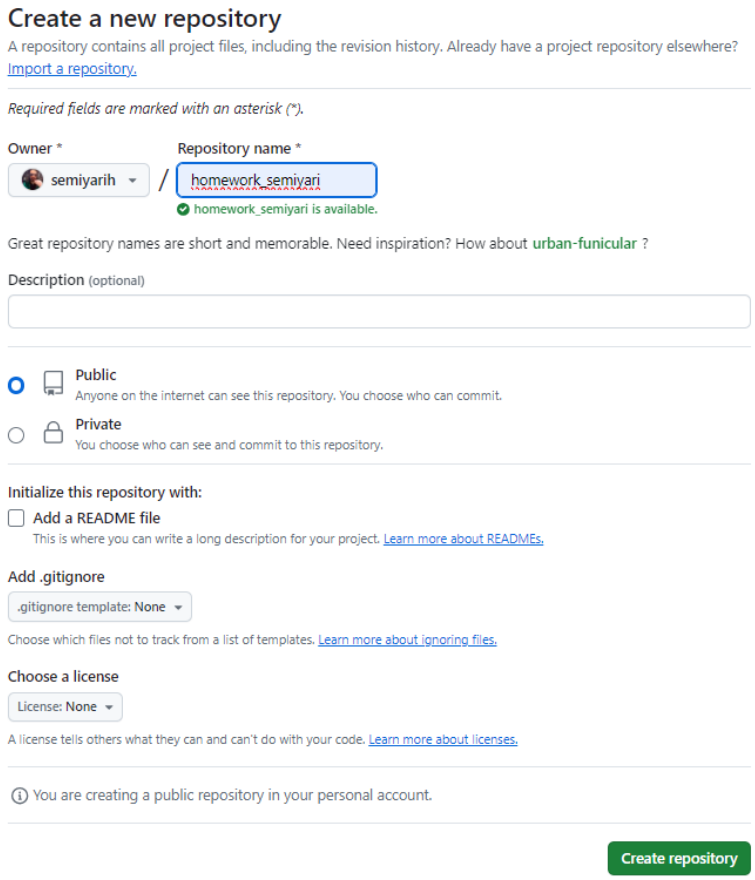
II. Add and Commit a File:

- Create a file, such as `README.md`.
- Add some content to `README.md`.
- Stage the file for commit: `git add README.md`.
- Commit the file: `git commit -m "Initial commit"`.



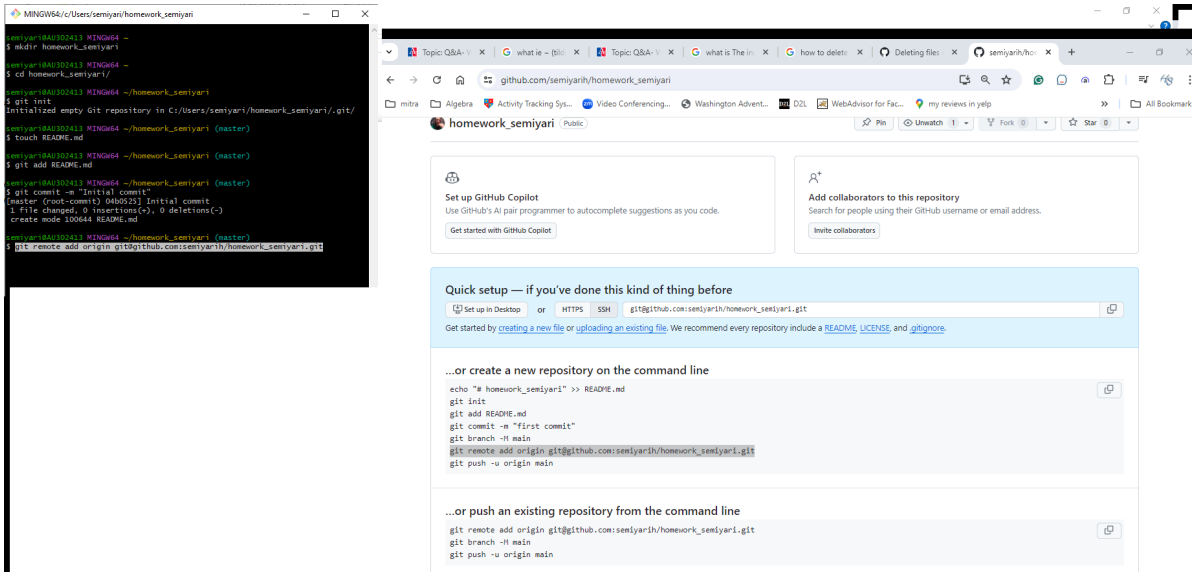
III. Create a Repository on GitHub:

- Log in to GitHub and create a new repository with the same name (homework_semiyari)



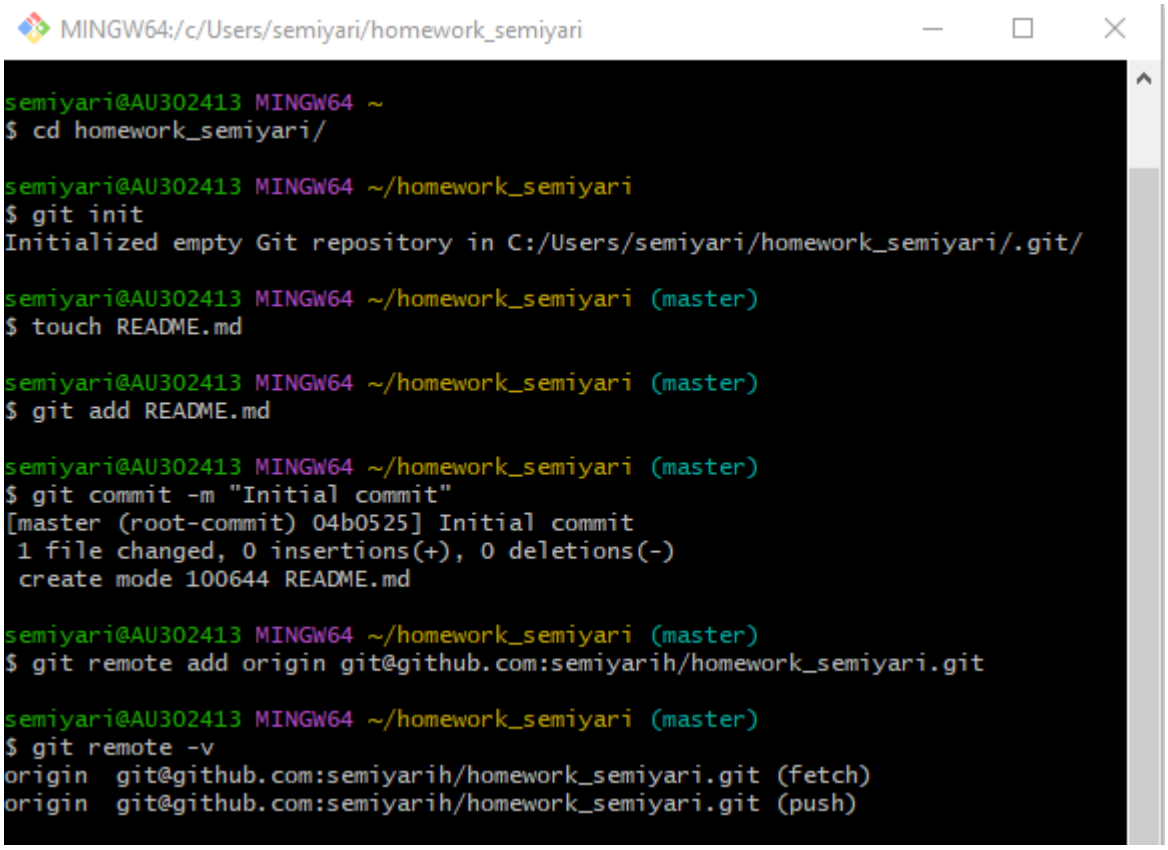
IV. Add GitHub Remote:

- Connect your local repository to GitHub by adding it as a remote



V. Verify Remote URL:

- Confirm the remote URL is set correctly: This command displays the URLs used for fetching and pushing to GitHub.
- This command displays the URLs used for fetching and pushing to GitHub.



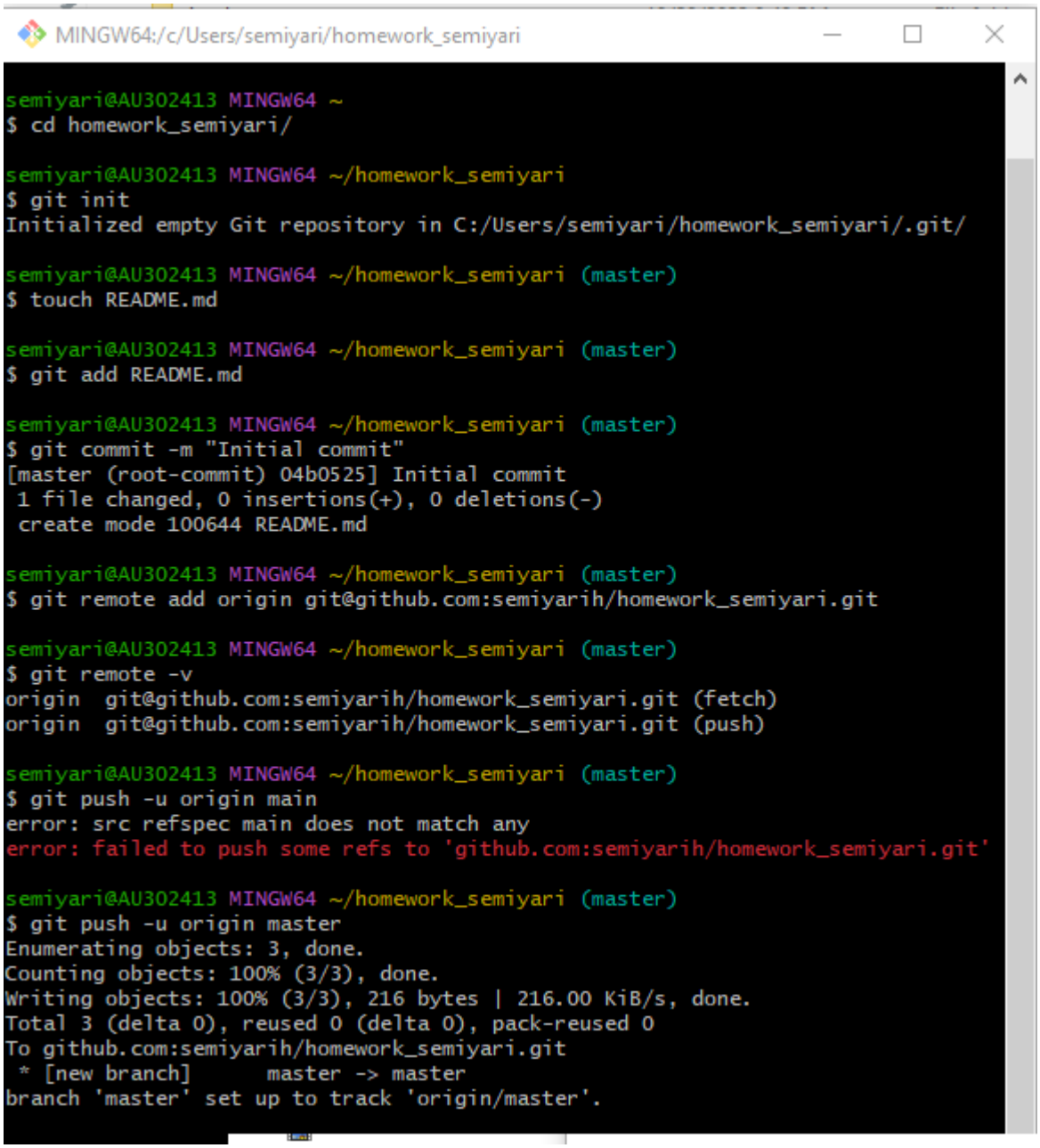
VI. Push to GitHub:

- Push your local repository to GitHub:

`` `git push -u origin main` ``

- Note:
  - Use `-u` for the first push to set the upstream branch.
  - If your local branch is named `master` instead of `main`, use:

```
git push -u origin master
```



```
MINGW64:/c/Users/semiyari/homework_semiyari

semiyari@AU302413 MINGW64 ~
$ cd homework_semiyari/

semiyari@AU302413 MINGW64 ~/homework_semiyari
$ git init
Initialized empty Git repository in C:/Users/semiyari/homework_semiyari/.git/

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ touch README.md

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git add README.md

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git commit -m "Initial commit"
[master (root-commit) 04b0525] Initial commit
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 README.md

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git remote add origin git@github.com:semiyarih/homework_semiyari.git

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git remote -v
origin  git@github.com:semiyarih/homework_semiyari.git (fetch)
origin  git@github.com:semiyarih/homework_semiyari.git (push)

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git push -u origin main
error: src refspec main does not match any
error: failed to push some refs to 'github.com:semiyarih/homework_semiyari.git'

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git push -u origin master
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 216 bytes | 216.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To github.com:semiyarih/homework_semiyari.git
 * [new branch]      master -> master
branch 'master' set up to track 'origin/master'.
```

Alternatively, rename your local branch to `main` before pushing:

```
git branch -m master main
git push -u origin main
```

These steps ensure your local repository is correctly linked to GitHub and that your changes are pushed to the remote repository.

Once you've completed the steps, you may want to verify the changes on GitHub.

Next, move your file (for example: previous homework assignment,Assignment 1) into the homework_semiyari (use your own last name) directory on your local machine. Then, stage, commit, and push these changes to your GitHub repository.

Question

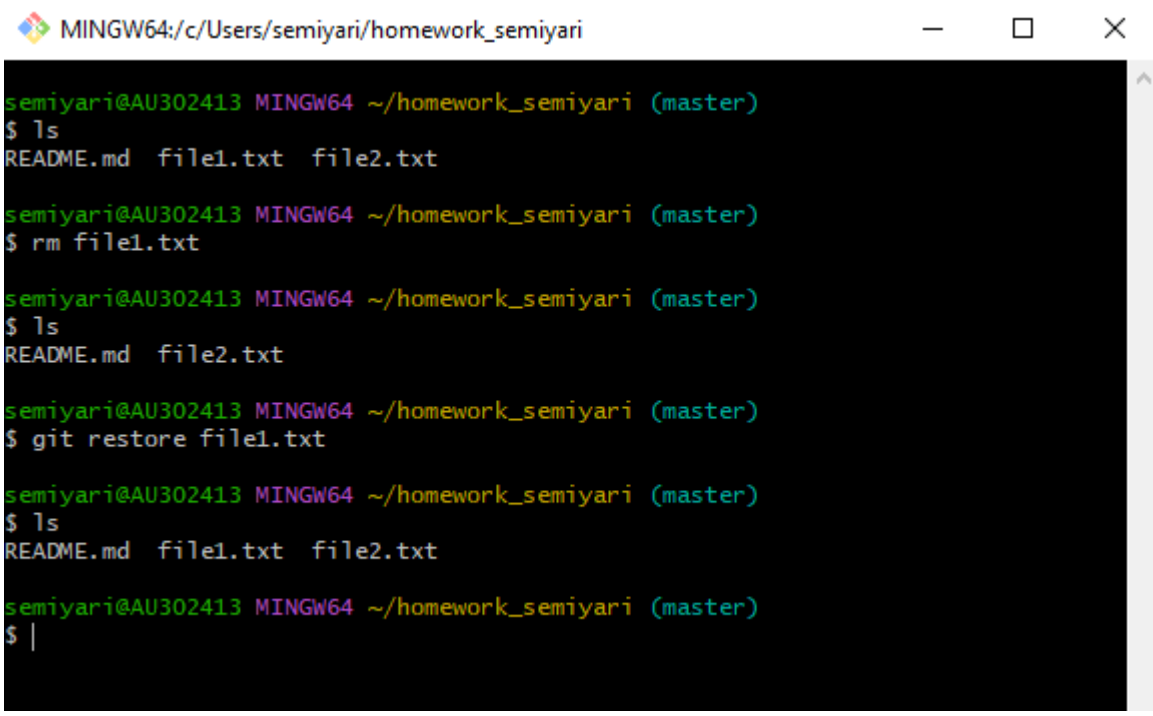
I accidentally deleted my files in my repo. How to recover it?

Recovering a Deleted File with Terminal/Bash

In my working directory, I added two files (`file1.txt` and `file2.txt`) and committed them one by one. Now, I want to delete `file1.txt` and then recover it.

There are three main cases for recovery:

1. **Just Deleted the File (Not Staged)** - If you have just deleted the file and have not staged it, type: `bash` `git restore file1.txt`



2. **Deleted and Staged the File** - If you have staged the file, meaning it was deleted and sent to the index area, type: `bash` `git restore --staged --worktree`

`file1.txt` - Explanation: - `--staged` tells Git to restore the file in the index from HEAD. - `--worktree` tells Git to restore the file in the working tree.

```
semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file1.txt  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ rm file1.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

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:    file1.txt

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

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git add file1.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git restore --staged --worktree file1.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file1.txt  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$
```

3. **Staged and Committed the File** - If you have staged and committed the file, you need to use `git checkout`. You also need the checksum (or hash) value of the deleted file.

There are various ways to find the hash value, and I have explained a couple of them in previous posts. Here is another way: `bash` `git rev-list HEAD -n 1 -- file1.txt` - `-n 1` tells Git to limit the result to only one commit.

- Once you have the checksum, you can use `git checkout` to retrieve the file. You do not need to write the entire checksum, just the first few characters (I usually use the first 6 characters, but you may type more or less). `bash` `git checkout 47d906^ file1.txt` - Explanation: - The `^` at the end of the commit hash tells Git to fetch the commit before this one.

MINGW64:/c/Users/semiyari/homework_semiyari

```
semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file1.txt  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ rm file1.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git add .

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git commit -m "Delete file1"
[master 47d906c] Delete file1
 1 file changed, 1 deletion(-)
 delete mode 100644 file1.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git status
On branch master
Your branch is ahead of 'origin/master' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$  git rev-list HEAD -n 1 -- file1.txt
47d906c786bfda64f207cdf53e92d82ed6e4fc28

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ git checkout 47d906^ file1.txt
Updated 1 path from 45e7d05

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$ ls
README.md  file1.txt  file2.txt

semiyari@AU302413 MINGW64 ~/homework_semiyari (master)
$
```

Question

I have an existing project on GitHub and want to change a few lines of code in an existing file. Or change a few lines in my README file

For editing `README.md` files or similar, it's best to use your preferred text editor. The terminal command varies depending on the editor.

For example, if you use "Visual Studio Code", you can open your README.md file by typing `code README.md` in the terminal. This command will open the file, allowing you to modify and save it.

Alternatively, you can navigate to the folder containing your `README.md` file and open it directly.

To modify an `R script` or `.QMD` file, open it in "RStudio", make your changes, and save the file.

After editing, you must `stage` and `commit` the file. *You should commit each time you change a file to ensure your repository stays updated.*

Question

What are the differences between git pull and git fetch. Also, what steps would I have to take to keep a forked repo up-to-date

Copying a repository can refer to either "forking" or "cloning," but these are different processes:

- **Forking a Repository:** Forking creates a copy of the original repository (called the upstream repository) on your GitHub account. This copy remains on GitHub and is not downloaded to your local machine.
- **Cloning a Repository:** Cloning creates a copy of the repository on your local machine using Git.

I. Forking a Repository To fork a repository, click the fork button in the upper right corner of the repository's page on GitHub.

II. Keeping a Forked Repository Up to Date To keep your forked repository up to date with the original (upstream) repository, you need to fetch updates from the upstream repository. You can also propose changes from your fork to the upstream repository using pull requests.

III. Using `git pull` to Update a Local Repository The `git pull` command updates the local version of a repository from a remote repository. This command combines two steps:

1. **Git Fetch:** Updates your remote-tracking branches with any changes from the upstream repository.
2. **Git Merge:** Merges the fetched commits into your local branch.

Running `git pull` fetches the remote repository and updates your currently checked-out branch with any new changes, ensuring your local repository is synchronized with the upstream repository.

Question

How do you create a new file? Does "git add _____" automatically create the file if none of that name exists?

There is a common confusion when first learning Git.

Creating a New File:

You create new files using your operating system's commands or your text editor.

For example, in bash, you can use,

```
touch newfile.txt
```

Alternatively, you can create a new file using a text editor like VSCode or Notepad++.

git add: The git add command is used to stage changes for committing. It does not create files.

Instead, it adds existing files or changes to the staging area.

For instance: `git add newfile.txt`

This command tells Git to track `newfile.txt` and prepares it to be committed. If `newfile.txt` does not exist, Git will throw an error.

I hope this helps clear things up!

Question

** When we add the README.md file, it provides a clear understanding of the project to new users. What are some tips for making a README.md file more engaging(allowing the user to easily apply or understand)?**

A README is often the first item a visitor will see when visiting your repository. README files typically include information on:

- What the project does
- Why the project is useful
- How users can get started with the project
- Where users can get help with your project
- Who maintains and contributes to the project

Question

I discovered that my GitHub repository not only have my files, but also a ".Rhistory file" . However, I haven't created a file called .Rhistory on my laptop yet. Where soes it come from?

The `.Rhistory` file is created by R to save the history of commands entered in the R console. This file is automatically generated when you use R and keeps a record of all commands executed during your R session. The `.Rhistory` file is typically stored in your working directory.

You can manually save the command history during an R session by using the `savehistory()` function. If no commands were executed during the session, the `.Rhistory` file might not be created, or if it was already created, it might not be updated.

If you don't see the `.Rhistory` file, ensure you are looking in the correct directory. It is also possible that the settings are configured not to save the history file automatically.

How to save history manually:

To save the history manually, type `savehistory("MyHistory")` (you may use any name for your history file).

How to view the history:

- **In RStudio:** RStudio has a History pane that automatically displays the command history.

Question

Differences between git reset and git revert. Understanding Git: `reset` vs `revert`

A branch pointer is a reference that points to the latest commit in a branch. A commit is like a snapshot of the entire repository at a specific moment. HEAD is a pointer to the most recent commit in the current branch.

Let's say we have a Git repository with this commit history:

```
A - B - C - D - E (HEAD)
```

We realize that we need to undo the changes made by commit `C`.

Using `git reset`

1. **`git reset --soft C`** - Moves the branch pointer and HEAD to commit `C`. - Keeps the changes from `D` and `E` in the staging area. - These changes are ready to be committed again if needed.

After the soft reset, the commit history looks like: `A - B - C (HEAD)` Changes from `D` and `E` are in the working directory and staging area but not in the commit history.

2. **`git reset C` (Default)** - Moves the branch pointer and HEAD to commit `C`. - Clears the staging area, but changes from `D` and `E` remain in the working directory.

After the mixed reset, the commit history looks like: `A - B - C (HEAD)` Changes from `D` and `E` are in the working directory but not staged.

3. **`git reset --hard C`** - Moves the branch pointer and HEAD to commit `C`. - Clears the staging area and discards all changes in the working directory. - Permanently deletes any changes made after commit `C`.

After the hard reset, the commit history looks like: `A - B - C (HEAD)` All changes from `D` and `E` are lost.

Using `git revert`

`git revert C` creates a new commit that undoes the changes made by commit `C` without changing the commit history.

After the revert, the commit history looks like:

```
A - B - C - D - E - F (HEAD)
```

Here, `F` is a new commit that undoes the changes made in commit `C`. The commit history is preserved, and the changes from commit `C` are reversed.

Question

a question about the content of the 8_git_github.pdf file. In the "Creating a New Branch" section, there is an image that contains "98ca9," "34ac2," and "f30ab." I'm curious about what these represent.

I noticed a question about the "Creating a New Branch" section in the document 8_git_github.pdf, specifically regarding the image displaying commit hashes like "98ca9," "34ac2," and "f30ab." These numbers are short representations of commit hashes used for illustrative purposes.

Please review the accompanying notes carefully, as the images serve as supplementary material and may require context to fully understand. For instance, if you refer to the screenshot of my terminal running the command `git log --oneline --decorate` on page 19, you'll see the hash value `db6dd67`, indicating the pointer references both the main and testing branches.

Note that the checksums (or hashes) you encounter will be unique to each commit and will differ for everyone.

Question:

Are the Hash value and Checksum the same?

While checksums and hashes are often used interchangeably, they have distinct meanings, especially in the context of Git. Checksums primarily verify data integrity, acting like fingerprints for data sets. On the other hand, a hash specifically identifies a unique commit in Git's history, serving as a pointer for tracking, referencing, and navigating commits.

To retrieve either the commit hash or checksum:

- Use `git log` or variations like `git log --oneline` to view commit history.
- To find the current full hash, use `git rev-parse HEAD`.
- For a shorter version of the hash, use `git rev-parse --short HEAD`.

In practice, both terms often refer to the same value. However, the distinction in terminology helps clarify their specific roles: "checksum" emphasizes data integrity and uniqueness, while "hash" specifically identifies a commit within Git's version control system.

Question:

What are the advantages and disadvantages of using SSH keys versus PATs?

Before August 2021, accessing repositories on platforms like GitHub was straightforward—you just needed a username and password. But for better security, newer methods like Personal Access Tokens (PATs) and SSH keys have become more common.

- PATs are tokens created by GitHub that act like a single key granting access, similar to a car key. However, because PATs rely on this single secret token, they are less secure compared to SSH keys.
- SSH keys work in pairs: a private key (kept safe on your device) and a public key (stored on GitHub). They use advanced encryption for high security. But setting up SSH keys can be complex, and they are tied to specific devices, which can be inconvenient if you need access from multiple devices.
- If you use PATs, you connect to your repository over HTTPS. In contrast, SSH connections use the Secure Shell (SSH) protocol, offering stronger security due to their cryptographic methods.
- In summary, SSH keys provide better security, while PATs are easier to use.

Check this website Authenticating with GitHub.

Question

Differences between `git rebase` and `git merge`, and share examples of when you would use one over the other?

Git – Difference Between Merging and Rebasing

The main difference between `git rebase` and `git merge` is in how they combine changes from different branches:

- **`git merge`**: Combines changes from two branches by creating a new merge commit that includes changes from both branches.
  - It merges two branches to create a "feature" branch.
  - It is more suitable for projects with the less active main branch and it is easier comapre to `rebasing`
  - It maintains the complete history of both branches.
  - It is suitable for projects with frequently active main branches.

  - It is preferable for large number of people working on a project.
  - It is preferable for small groups of people.
  - Example: Imagine you have two branches: `feature` and `main`. You want to merge the changes from `feature` into `main`.

Switch to the `main` branch

```
git checkout main
```

Merge the `feature` branch into the `main` branch

```
git merge feature
```

- **`git rebase`**: Takes your commits from one branch and applies them on top of another branch, creating new commits.
  - rebases the feature branch to add the feature branch to the main branch.
  - It Doesn't maintain the history of both branches.
  - Example

Imagine you have two branches: `feature` and `main`. You want to rebase `feature` onto `main`.

Switch to the `feature` branch

```
git checkout feature
```

Rebase the `feature` branch onto the `main` branch

```
git rebase main
```

After rebasing, you may need to force `push` the changes if the branch is shared:

```
git push origin feature --force
```

Both commands let you combine branches in Git.

Question

Is there any way to handle such merge conflicts in Git when multiple branches have breaking and overlapping changes?

Git can automatically handle merges in most cases, but conflicts occur when two branches edit the same line or when a file is deleted in one branch but edited in another. These conflicts are common in team environments.

If multiple branches encounter conflicts, Git's merge process becomes more complex and may require manual resolution.

Resolving a merge conflict using the command line