HW01: Editing Files, Safe Git Collaboration, & Daily Workflow

Overview

Your course project involves you working on pieces of a larger task, something that cannot be completed in a few weeks. The project requires you to accomplish goals weekly. As part of your weekly task, at very least, you will back up your weekly coding contributions on GitHub. On some occasions, you will integrate your contributions with those from other collaborators. This homework starts this process and also walks you through some safe collaboration methods for a team collaborating on an RStudio project connected to a remote GitHub repo.

This homework file is lengthy in order to walk you through the process of using Git and collaborating with others in a version-control environment. The coding elements, however, are lean.

Homework Deliverable:

For this homework, you will work on independent elements of the project. You will stage, commit, and push file edits on your personal branch as well as merge the `dev` branch into your branch. Your homework submission will be your modified files and commit history on your personal branch of your team's GitHub remote.

You will need to ensure that you also have your personal feature branch configured as done in a previous exercise. Please review **the team-project-git-exercise.pdf**.

If you need assistance on reading or writing files, please review the **R Basics & R Refresher on the course page**.

Things you will need:

- Your team GitHub repository and an RStudio Git Version Control project connected to that remote repo
- The project opened in RStudio (e.g., click `<project-name>.Rproj` to open)

Unless you are a skilled Git user, I do not recommend diverging from the guidance approach or you may find yourself troubleshooting a Git conflicts or errors.

The Project Directory Structure:

Your project directory is structured as seen below. By design, your team project also contains sample files for this homework. Those files interact with the `mpg` data set (built into R) so that you familiarize yourself with working with files an R project. You will extend the experience to your own project's files.

```
.
|-- _quarto.yml                      # Quarto project configuration file
|-- data/                            # Data files
|   |-- raw/                         # Original
|   |-- processed/                   # Cleaned
|-- src/                             # R scripts
|   |-- data/                        # Data cleaning/processing scripts
|   |   |-- clean_mpg_data.R
|   |-- figs/                        # Plotting scripts
|   |   |-- plot_cty_hwy_mpg_point.R
|-- figs/                            # Saved plots
|   |   |-- cty_hwy_mpg_point.png
|-- pages/                           # Quarto website pages
|   |-- team_test_page.qmd
|-- docs/                            # Rendered website files
```

Project Branches and General Weekly Workflow:

Branches: a `main` branch, a `dev` branch for development/testing, and your personal feature branch both locally and on the remote.

Weekly Workflow:

- Each collaborator **develops code on their personal feature branch**, `<my-branch-name>`.
- Collaborators push their changes to their remote regularly (after finishing a task/day).
- Before merging personal changes into `dev` collaborators **first perform a local merge of `dev` into their personal branch** in order to troubleshoot conflicts in an unshared branch.

Collaborator Homework Tasks and Responsibilities:

In alphabetical order based on *Last* name, the member with the last name earliest in the alphabet will be Collaborator 1 and so forth until Collaborator 4. You do not have to wait for your collaborators to complete your task.

- **Collaborator 1**: Create/modify an `.R` script to clean the data (`src/data/clean_mpg_data.R`)
- **Collaborator 2**: Create/modify an `.R` script to create and save a plot of the data (`src/figs/plot_cty_hwy_mpg_point.R`) using the cleaned data file (`data/processed/cleaned_mpg.Rds`)
- **Collaborator 3**: Create/modify a Quarto webpage file (`pages/team_test_page.qmd`) to source `src/data/clean_mpg_data.R` and `plot_cty_hwy_mpg_point.R` to clean and plot the data; and call the `.png` file for the plot referencing a saved png figure (`figs/cty_hwy_mpg_point.png`)
- **Collaborator 4**: Create a version of/modify an `.R` template script (`src/starter_script_file.R`)

Homework Part A: Coding Contributions to the Project

This is where the fun begins. In order to collaborate, you need to code. After ensuring that you are on your personal branch (e.g., `git switch`), start your file edits in RStudio. From the Environment pane, select the *Files* tab (not the *File* tab from the Source pane) and navigate to your designated file.

Although you will be creating your own project code, this homework requires editing files already provided for you.

Collaborator 1: Reading, Cleaning, and Writing Data

In RStudio, modify `src/data/clean_mpg_data.R` in order to:

- Read the raw data file `data/raw/mpg.Rds`
- Clean the raw data using `{dplyr}`
- Write the cleaned file `data/processed/cleaned_mpg.Rds`
- Ensure that your data file is present
- Save your `.R` script

Collaborator 2: Plotting Data and Saving Plots

The cleaned data from Collaborator #1's coding is already available to you, allowing you to work on your part. This scenario will **not** be the case when working on a real team project as your teammates will need to make those available on the `dev` branch.

In RStudio, modify `src/figs/plot_cty_hwy_mpg_point.R` in order to:

- Read the *cleaned* data file `data/processed/cleaned_mpg.Rds` (based on Collaborator #1's efforts)
- Plot the data using `{ggplot2}`
- Save the plot `figs/cty_hwy_mpg_point.png`
- Ensure that your plot file is present
- Save your `.R` script

Collaborator 3: Editing a Quarto Webpage File

The cleaned data and plot created by Collaborators #1 and #2 is available to you. This scenario will **not** be the case when working on a real team project as your collaborators will need to make those edits available on the `dev` branch.

Every time your webpage is built, you will need to ensure the most up-to-date and accurate file is used to render the page files. This means that reading and cleaning data as well as plotting and saving plots is performed at this final step.

In RStudio, modify `pages/team_test_page.qmd` in order to:

- Call/source Collaborator #1's cleaning script, `src/data/clean_mpg_data.R`
- Call/source Collaborator #2's plotting script, `src/figs/plot_cty_hwy_mpg_point.R`
- Display the most recent plot `.png` file, `figs/cty_hwy_mpg_point.png`
- Save your `.qmd` file

Collaborator 4: Understanding the `starter_script.R` template

Having clean and clear script files makes for a tidy project. The `src/starter_script_file.R` will serve as a starter script for all script coding on the team. When you have new data cleaning, new plotting, or a new function, all team members will start with this file because it is constructed to remind you of keeping your code tidy. Open `src/starter_script_file.R` and save a new version with the name `using_starter_script.R` so that you do not overwrite the original. You should save it in `src/data/`. If you do not save the file with a new name, your starter script may be forever changed.

Atop the file, you will see details for naming the script, an author name, a GitHub account, a date, and a purpose for the script. Practice entering in details so that you can later tell your teammates how to ensure you all complete this and take authorship of your own files. You will see that code blocks are included, where you can easily place code for loading libraries, reading data, processing data, etc. in their own sections.

When you are done, save your `.R` script. Your task is less technical others in the homework. Consider looking at Collaborator #2's task and *think about* how you might handle the task yourself.

Homework Part B: Backing up your Local Branch

At this point, you have edited your personal files *locally* on your personal branch. There are no new changes on `origin` (the remote) because you and your collaborators have not yet pushed changes to the remote. Nevertheless, to familiarize you with the day-to-day workflow, you will practice certain steps as if changes from collaborators have been pushed to the remote.

Step 1: Stage/Add

Having made recent file edits, you will make these changes accessible on `origin` as a backup to your local branch just in case you overwrite them and so that collaborators will have access to your changes.

Key Files:

- **Collaborator 1** modified: `src/data/clean_mpg_data.R`
- **Collaborator 2** modified: `src/figs/plot_cty_hwy_mpg_point.R`
- **Collaborator 3** modified: `pages/team_test_page.qmd`
- **Collaborator 4** modified: `src/data/using_starter_script.R`

Important: Only `add` and `commit` files that **you have created or modified** on your personal branch. Never commit other collaborators' files. Your authorship provided in your scripts will make this clear but you may need **a way to distinguish file names** so that you manage them easily.

You will now *add* files in a targeted manner using the path and file name.

```
git add <path-to-file-and-filename.R>
```

Pro Tip: Depending on your Git configuration, after typing some characters to your file, you may be able to use TAB to partially auto-complete the file path. However, always ensure you are adding the correct file as auto-complete may result in inaccurate file selection.

WARNING: Don't use `git add .` to add files. This approach will prevent you from staging files you do not intend to stage. Moreover, `git add .` can accidentally stage Quarto cache files, `.Rproj.user` files, temporary output files, or other files that you do not wish to be part of a commit. Always add files explicitly by file name.

Step 2: Commit your Staged Files

You have added a file change. You will now tack on a message describing what you worked on so that your future self and others can understand the edits you made. This is referred to as adding a commit message. *Commit* your change with a message using `commit -m` and your message in either single or double quotes:

```
git commit -m "xyz script for ..."
```

Step 3: Push File Changes

Push your local branch to `origin` on GitHub

```
git push origin <my-branch-name>
```

This step will ensure that your remote branch is up-to-date with your local branch. Inform your teammates of your new completed work.

Homework Part C: Managing File Paths Appropriately

Step 1: Evaluating Path Management

In order to ensure that your project renders, your paths to all directories and files need to exist on the platform running the code. How did your team decide to manage and code file paths? Did all team members have different paths to files because they have different user names on their computers? Did you collaborate with people who have the same name to make things easy?

Did you hard-code the paths like these?

`"/usr/beavis/desktop/this_project_dir/src/figs/plot_cty_hwy_mpg_point.R"` or `"C:/Users/beavis/desktop/this_project_dir/src/figs/plot_cty_hwy_mpg_point.R"`

Hard-coding file paths will not allow you to run the code without changing all of the file paths on each user's system. Moreover, what will happen if or when you move the project directory someplace on your computer? Will you need to change all of the paths again?

Step 2: Managing File Paths with `{here}`

Test the `here::here()` or `here()` function:

```
here::here()
```

Notice the returned path is the root to the RStudio project directory, always and forever. But how do you build a path to a file located in a sub-directory of the project?

Read the function docs to understand what it is doing.

```
?here::here
```

You will read that you need to build a file path with character arguments.

For example:

```
here::here("dir", "anotherdir", "my_file.R")
```

You will read that a directory path is built with a `/` separating each argument you pass to the function.

Just because you build the path, does not mean it actually exists on your file system. Check whether the path exists using `fs::file_exists()`. Unless the file (or directory) exists, the path is useless.

```
fs::file_exists(here::here("dir", "anotherdir", "etc"))
```

Pro Tip: `{fs}` is a much better library for managing files on your *f*ile *s*ystem than the `{base}` library. You could have used `file.exists()` but I recommend using and becoming comfortable with `{fs}` as doing so will open more doors for you. For directories, the `{fs}` equivalent to `dir.exists()` is `fs::dir_exists()`.

Build a proper path to a file you believe exists and check whether it exists:

```
fs::file_exists(here::here("src", "figs", "plot_cty_hwy_mpg_point.R"))
```

No matter which system you are on, `{here}` will manage your file and directory paths **without hard coding**. This is the approach you will take for **all** project paths.

Step 3: Fixing File Paths with `{here}`

If you did not use `{here}` for path management, you will need to fix your file paths and repeat the steps associated with editing files and pushing your changes to the remote.

Back up your branch again by staging, committing, and pushing your file change if you did not use {here} when building your file path(s).

Parts A, B, and C are required for the homework. Your project will demand more from you, however, as you collaborate with others. You will need to obtain changes from others in order to carry out elements of your own work or you will need to integrate elements into a larger composite project. I really recommend you not delay familiarizing yourself with the workflow process. Testing this week will offer benefits. I may also offer bonus to those who work through Parts D and E. Either way, you will need to do it.

Bonus Part D: Understanding Typical Daily Workflow: Version Control with Git and GitHub

You will need to test your changes alongside other changes will inevitably take place. A safe way to test changes is to merge your collaborators' changes with your own changes. In order to do so, you need to ensure that your local files reflect changes on a remote branch.

Step 1: Update your local `dev` Branch

Switch to the `dev` branch, fetch changes, and pull them from the remote to your local `dev`.

Switch to local `dev`:

```
git switch dev
```

Pull remote `dev` changes on origin to the local `dev`:

```
git pull origin dev
```

Step 2: Merge Updates into your Personal Branch

Now that you fetched those changes from the remote and they exist locally, you can merge them from `dev` to your personal branch. However, because you switched to `dev` to fetch the changes, you will need to **`switch` to your personal branch** so that you can merge those updates into your personal branch, `beavis`.

```
git switch beavis
```

Step 3: Merge `dev` into your Personal Local Branch

```
git merge dev
```

Because you switched to your personal branch, this command merges the file changes from `dev` into your personal branch, locally anyway. If there were changes integrated, your local branch will be "ahead" of your remote branch.

After merging `dev` into your personal branch, your local personal branch will include new commits from `dev`.

If the merge was clean (no conflicts), just push to update your remote with the changes. No need to add or commit anything. There should not be conflicts at this stage if you are working on and pushing your own files.

```
git push origin beavis
```

Note: As long as you are working in separate files and not working in the same file, there should be no conflicts. Managing conflicts in your personal branch is safer than managing them in a shared branch like `dev`. This approach also keeps responsibility and conflicts localized, taking a *you break it, you fix the issue in your own branch before merging.* There are ways to merge a collaborator's branch into your own branch (e.g., `git merge origin/<collaborator-branch-name>`) but this makes someone's issues now your responsibility. Such an approach would best be managed by a project coding lead.

Step 4: Fetch and Merge your Personal Branch into `dev`

When you are ready to share your work (at the end of a day or week) and have already staged, committed, pushed your edits to your remote, and you have ensured that the `dev` branch does not cause problems for your branch, you can integrate those changes into the `dev` branch.

I advise also that you communicate with your teammates that you are doing this so that you **don't all complete this step at the same time**. Performing a merge in steps ensures there are either no conflicts or that you know from where the conflicts originated. One of you should do this at a time, perhaps dictated by a code lead or co-code lead.

1. Ensure `dev` is up to date by pulling any possible changes from `origin`.

For best results, always pull the most recent changes. In other words, never trust that collaborators have not just also made changes.

```
git switch dev
git pull origin dev
```

You have the most recent, if any, changes to `dev` on `origin`.

2. Merge your local personal branch into your local `dev` branch.

Having immediately pushed your changes to your remote, merge them into `dev`.

```
git merge beavis
```

3. Push your local `dev` branch to `origin`.

To ensure that the merge result is also in sync with the remote, push.

```
git push origin dev
```

Pro Tip: Always ensure your local branch and remote are in sync. If you push your personal branch to the remote, in a matter of seconds both your local and remote branches will be identical. Therefore, merging your local personal into the local `dev` ensures what is merged is also the the remote. If, however, you subsequently make other edits and stage and commit them, your local branch will be ahead of your remote branch by that one commit. If you do not push your branch to the remote, merging your local branch will similarly result in your `dev` branch containing changes that your remote branch is lacking.

(Final) Bonus Part E: Rendering the Website Pages

Now that you have all of your collaborators files, and your paths are managed, you should all be able to render the website. All team members will be able to build the website but only with integration of your work will it take the desired form.

The easiest way to render the website is to select from the toolbar `Build > Render Website`.

Ensure that you have the `{quarto}` library installed if you have not yet installed it. When you render the website, the project will build into `docs/` and you should be prompted to open the website in a browser.

Later, you will need to push `docs/` to the remote and set up GitHub Pages for serving up the pages but that task is for another day.

A Note on sourcing code in `.qmd` Files

All collaborators' code represents a feature of a much larger project. Everyone needs to ensure their scripts work on their own but also when sourced by other scripts. Because you will be regularly merging your `dev` into your personal branch, everything you need to render the project on your own will be in your personal branch and ready for test rendering.

Pro Tip: After making edits to files that are integrated with other code, and especially `.qmd` files, you could perform a re-render to ensure that your code did not break the rendering process.
