Exercise: Project Management

This class exercise will get you started with having an organized project. You will create a clearly documented script file, stage the file, commit it, and push it to a remote repository on GitHub. The workflow will allow you to be organized and clear and collaborate effectively with team members. If you are unable to complete the exercise in class, I encourage you to work through it on your own so that you understand the steps moving forward. The information covered depends on having completed the associated assignments by class time.

What is Covered/Provided

- {here} for managing directory and file paths
- Creating/Using RStudio projects with version control
- A starter script for clarity of script documentation
- A date file for the exercise
- Git commands for communicating with the remote repo

Understanding the `{here}` Library

This library is used for path management. When you want to reference the path to either a directory or a file, this library will serve you well. The `here()` function from the library will be used to return the full directory path to the RStudio Project as well as to build paths.

Example:

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

```
## [1] "C:/Users/gcook/Sync/git/dataviz/dataviz24"
```

Adding string arguments (e.g., single or double quotes) will extend that path to build a desired directory path. When adding arguments to the function, each argument is separated by a `/`. What is returned is a character string.

```r
here::here("data")
```

```
## [1] "C:/Users/gcook/Sync/git/dataviz/dataviz24/data"
```

```r
here::here("data", "raw")
```

```
## [1] "C:/Users/gcook/Sync/git/dataviz/dataviz24/data/raw"
```

There is no guarantee that the string represents an existing directory path. You must ensure on your own that the string represents a true file path exists using functions like `dir.exists()` or `file.exists()`. If you wish to understand a the function, precede its name with a ? (e.g., `?file.exists`) or use the `help()` function (e.g., `help(file.exists)`) and type it into the R console.

```r
here::here("data", "thisdirectorydoesnotexist")
```

```
## [1] "C:/Users/gcook/Sync/git/dataviz/dataviz24/data/thisdirectorydoesnotexist"
```

When the path is used to build a file path, the last argument will be the file name along with the file extension (e.g., `.csv`, `.Rds`, `.R`).

```r
here::here("data", "raw", "file-name.csv")
```

```
## [1] "C:/Users/gcook/Sync/git/dataviz/dataviz24/data/raw/file-name.csv"
```

Create an RStudio Project with Git Version Control

Step 1: Create a Git Version Controlled RStudio project

- Create a New Project, Verion Control, Git
- Set the repository URL to: `https://github.com/slicesofdata/dataviz24-<your-liaison-name>`
- Name the project name to `dataviz24-<your-liaison-name>`
- Specify the directory as your `/dataviz` class directory. If you have not created the directory, go back and read the modules on project management.

Step 2: Wait for the remote repository on GitHub to be cloned on your local computer. The `main` branch of this repo will be *pulled* from GitHub.

Step 4: Take inventory of the directories in the project.

- Make a note of the `/data` directory and sub-directories for data files
- Make a note of the `/src` directory and sub-directories for source code
- Make a note of the `/figs` directory for plot files
- Make a note of any files in those directories

I assume you have read about the project directory structure. Make sure that you know were to save files for your project so that it is well managed. If you save files in the incorrect places, your Code Lead will have you fix them.

Using Git: Your Own Feature Branch

A Git repository is like a tree. There is a main trunk as well as branches. There are different types of branching methods in Git. One is often referred to as *feature branching*, which is used to create a feature of a project. You can think of your code and plot file contributions to the project as a feature or set of features of the bigger project.

Step 1: Checkout a *Feature* branch

In your Git terminal create and checkout a feature branch using:

```
git checkout -b name-of-feature-branch
```

Example: `git checkout -b beavis` (if your name is Beavis)

You now need to create work (e.g., files) so that you can stage (add) them, commit them, and push them to the repository.

Step 2: Setting the Upstream Branch

When you create a new branch, you cannot `git push` to it until you have set an upstream branch on the remote repository. For the first push, you will need a special command. After setting the upstream, you can use `git push` when on that feature branch.

```
git push --set-upstream origin your-feature-branch-name
```

Creating R Code Script Files

Arrange in groups of 3-4.

Person A:

- Open `starter_script_file.R` from the `src/` directory
- Save a new version named `read_data_babynames.R` and save to the `src/data/` directory
- Inside the script file, write R code to read `baby-names-2008.csv` from `data/raw/` using the `read.csv()` function. Set the argument to the `file` parameter using `here::here()`

Example: `read.csv(file = here::here(<your-arguments>)`

- Inside the script file, write R code to write the data frame as an `.Rds` file using the `saveRDS()` function. Using `here::here()` set the argument to the `file` parameter so that the `.Rds` file is written to `/data/processed` with a new name, `"baby-names-2008-cleaned.Rds"`.
- Save your script file. Check that your file is in `/scr/data`.
- Check the `/data/processed` directory for your `.Rds` plot file.

Person B:

- Open `starter_script_file.R` from the `src/` directory
- Save a new version named `plot_baby_hist.R` and save to the `src/figs/` directory
- In the script, add a section to load the `{ggplot2}` library using the code below

```r
library(ggplot2)
```

- In the script, add a section using the code below for reading data using the `here::here()` function.

```r
dat <- read.csv(file = here::here("data", "raw", "baby-names-2008.csv"))
```

- Add a section for filtering the data, creating a plot, and saving it as a `.png` file. When saving, use the `here::here()` function to set the argument to `filename` so that the image file is written to `/figs` (not `/src/figs` as `/src` directories are used for source code) and with the name `"baby_hist.png"`.

```r
baby_hist <-
  dat |>
  ggplot(mapping = aes(x = percent)) +
  geom_histogram() +
  ggtitle("Histogram for 2008")

ggsave(filename = here::here(<your-argument>),
       plot = baby_hist,
       device = "png"
       )
```

- Save your script file. Check that your file is in `/scr/figs`.
- Check the `/figs` directory for your `.png` plot file.

Person C:

- Open `starter_script_file.R` from the `src/` directory
- Save a new version named `plot_baby_subset_col.R` and save to the `src/figs/` directory
- In the script, add a section to load the `{ggplot2}` library using the code below

```r
library(ggplot2)
```

- In the script, add a section using the code below for reading data using the `here::here()` function

```r
dat <- read.csv(file = here::here("data", "raw", "baby-names-2008.csv"))
```

- Add a section for filtering the data, creating a plot, and saving it as a `.png` file. When saving, use the `here::here()` function to set the argument to `filename` so that the image file is written to `/figs` (not `/src/figs` as `/src` if for source code) and with the name `"baby_subset_col.png"`.

```r
baby_subset_col <-
  dat |>
  filter(name %in% c("Jacob", "Emma", "Damari", "Kenley")) |>
  ggplot(mapping = aes(x = name,
                       y = percent
                       )
         ) +
  geom_col()

# if you want, examine the plot object

ggsave(filename = here::here(<your-argument>),
       plot = baby_subset_col,
       device = "png"
       )
```

- Save your script file. Check that your file is in `/scr/figs`.
- Check the `/figs` directory for your `.png` plot file.

Person D:

- Open `starter_script_file.R` from the `src/` directory
- Save a new version named `plot_baby_subset_point_line.R` and save to the `src/figs/` directory
- In the script, add a section to load the `{ggplot2}` library using the code below

```r
library(ggplot2)
```

- In the script, add a section using the code below for reading data using the `here::here()` function

```r
dat <- read.csv(file = here::here("data", "raw", "baby-names-2008.csv"))
```

- Add a section for filtering the data, creating a plot, and saving it as a `.png` file. When saving, use the `here::here()` function to set the argument to `filename` so that the image file is written to `/figs` (not `/src/figs` as `/src` directories are used for source code) and with the name `"baby_subset_point_line.png"`.

```r
dat <- read.csv(file = here::here(<your-arguments>))
```

- Add a section for filtering the data, creating a plot, and saving it as a `.png` file. When saving, use the `here::here()` function to set the argument to `filename` so that the image file is written to `/figs` (not `/src/figs` as `/src` directories are used for source code) and with the name `"baby_subset_point_line.png"`.

```r
baby_subset_point_line <-
  dat |>
  filter(name %in% c("Jacob", "Emma", "Damari", "Kenley")) |>
  ggplot(mapping = aes(x = name,
                       y = percent,
                       group = 1
                       )
         ) +
  geom_point() +
  geom_line()

# if you want, examine the plot object
```

```
ggsave(filename = here::here(<your-arguments>),
       plot = baby_subset_point_line,
       device = "png"
       )
```

- Save your script file. Check that your file is in `/scr/figs`.
- Check the `/figs` directory for your .png plot file.

Staging, Committing, and Pushing your Work to the Remote

Once you are certain that a file in your feature branch is ready for production or ready for others to use, you will want to `stage` it, `commit` it, and `push` it to the remote repository

Step 1: Staging

Examples depending on files you created:

Person A:

- `git add src/data/read_data_babynames.R`
- `git add data/processed/baby-names-2008-cleaned.Rds`

Persons B, C, or D:

- `git add scr/figs/plot_<your-plot-type>.R`
- `git add figs/plot_baby_<plot-type>.png`

Step 2: Committing

Commit your staged file(s) along with a message. You will need to add `-m` and a message inside quotes. Do not forget the quotes.

Person A:

- `git commit -m "added files for reading raw baby names and output rds file"`

Persons B, C, or D:

- `git commit -m "added files for making plot for baby names and output png file"`

Step 3: Pushing

As long as you have set your upstream branch, you can use `git push` to push your changes to the remote repository.

```
git push
```

Pull Request

A *pull request* is used for developers to notify collaborators that you have completed a feature or a set of changes in a separate branch of the repository and you would like those changes to be reviewed and potentially merged into a main branch. This request is often used for code review for catching bugs or making modification suggestions. The request also serves to document the changes. Once the code is approved, it can be merged into the main branch.

Step 1: Click the link for the pull request

Immediately after your file(s) are pushed to the remote, you will see a message about a *pull request* in your terminal. The message will include a link.

For example, you will see a message like:

```
remote: Create a pull request for 'your-feature-branch-name' on GitHub by visiting:
remote:      https://github.com/<github-user>/<repo-name>/pull/new/<your-feature-branch-name>
```

Once you click the link, GitHub.com should open in your browser and you will have the option to perform the pull request, verify there are no conflicts and perform the pull request.

Step 2: Open a pull request

A page will open to *Open a pull request.* You can add a title and a description of what you have pushed so that there is clarity for your future self and your collaborators. Once satisfied, click **create pull request**.

Step 2: Open a pull request

GitHub will check for conflicts and let you know if there are any. If you are performing your own work with your own files and your file names do not conflict with file names created by team members, you should not see any conflicts.

Merging your Feature Branch with the Main Branch

After the pull request, you will want to merge your files (from your feature branch) into the main branch so that team members can use those files or so that the Code Lead can routinely check the developing project. If you are certain all is OK, you may be able to do this yourself.

Step 1: Merge Pull Request

Your browser will load a page that start with something like **added new file** along with the description from the pull request (above). A box will also inform you whether there are any conflicts with merging and if not, you should see green check mark along with the message that reads *This branch has no conflicts with the base branch*.

Click the option to **merge pull request**.

Step 2: Confirm Merge

After the merge, there will be an option to *confirm* the merge.

Click **confirm merge** to confirm. The page should be updated with the message *Pull request successfully merged and closed*.

Note: The final project will require that all features are merged into the main branch. More likely, a feature you code may be needed for other team members to do their work or similarly, your work may be dependent on their feature(s). When you push changes that are needed by you team, let them know so that they can checkout the main branch and pull the recent changes. If the changes are not, needed, then you do not need to alert them.

Obtaining the Work of Others

Pushing files to the remote using `git push` stores the files on GitHub. Because you created those files locally yourself, they are available in your local git repo. Files pushed to the remote repository by your team members will be seen on GitHub but they will be unavailable to you in your local repo. We need to get them.

Although you can pull changes from the main branch without leaving a feature branch, newbies may make mistakes. As such, we will use the *traditional* approach which represents a standard sequence of Git commands used to update the main branch with the latest changes from the remote repository and then either switch back to an existing branch or create a new branch. Of course, with this approach, you will want to ensure you checkout a feature branch right away so that you are not on the main branch.

Step 1: Check Out the Main Branch

```
git checkout main
```

Step 2: Pull the Latest Changes

To ensure that you are pulling the changes from the main branch, use:

```
git pull origin main
```

Warning: Depending on configuration, `git pull` alone may not ensure pulling of correct files.

Step 3: Switch Off Main Branch

Because you want to be working on your branch or a feature branch and not the main branch, you will want to checkout your existing feature branch or create a new one for a new task.

To switch to your existing feature branch:

```
git checkout name-of-existing-feature-branch
```

To checkout a new feature branch, use `-b` to create branch and checkout.

To checkout a new feature branch, you will need something slightly different:

```
git checkout -b name-of-new-feature-branch
```

Step 4: Admire your team's work