Exercise: Grouping, Summarizing, and Plotting

Overview

This exercises addresses reading and writing data, aggregating data, managing files, and creating plots with multiple layers and different data sets. This exercises is designed to be completed in 75 or fewer minutes. Although you should have time to complete parts, as with all exercises, if you do not finish, I encourage you to complete it outside of class time so that you are able to incorporate your experiences and knowledge into future exercises. You may need to consult course reading materials located at the course site as some elements may not have been covered in the basic content contained in associated videos.

This exercise assumes you have an understanding of `{dplyr}` functions like `select()`, `filter()`, `mutate()`, and `summarize()` and that you have been practicing using these functions as part of your course allocation time outside of class. Also assumed is a basic understanding of the `{ggplot2}`'s `ggplot()`, `geom_point()`, `aes()` functions as well as conceptual plot-layering process implemented with `+` between layers rather than `|>`.

This exercise focuses on:

- Intentional File Creation and File Management
- Data Aggregation
- Sourcing Script Files
- Plot Layering Using Different Data Frames
- Workflow and Reproducibility
- Version control

This exercise uses:

- Your RStudio Git version-control `dataviz-exercises` project
- `{here}`, `{dplyr}`, `{ggplot2}` functions and functions from relevant Base R libraries (e.g., `readRDS()`, `saveRDS()`, `read.csv()`, etc.)
- Your knowledge from past exercises, videos, homework, etc.
- Data from `ggplot2::diamonds`

Collaborating

I encourage you to collaborate with and help one another but I will not assemble you into groups. Organizing by teams would be a great idea.

Getting Data

I encourage you to load the `ggplot2::diamonds` data and create your own `.Rds` file to use for this exercise, however, you can download a version from the Data Tab on course site and save in your `data/raw/` directory.

Part 1: Reading, Aggregating, and Writing Data

Consider data approaches that you might use to aggregate data for your team project (e.g., count/tally, sum, cumulative sum, mean, median, minimum values, maximum values, quantiles, variance metrics, etc.). Consider also relevant grouping variables for that aggregation procedure. If you apply these approaches to this exercise, you can reapply them to your team project.

1. Create an `.R` script file (Recommendation: Start with `starter_script_file.R`)

2. Name the file something like `aggregate_<var-name>_by_<grouping-var-name>.R` so the file name unambiguously describes your data aggregation goal. Use lowercase letters only; do not use spaces. Fill out authorship and description details, etc. Consider adding either your name or your initials as a prefix to the file name if your team decided to take that approach.

3. In a "library" section of your script, load libraries you need/use (don't load unnecessary ones).

4. In a "data-reading" section of your script, write your code to read `ggplot2_diamonds.Rds` using `{here}` paths. Assign the data frame an object name that unambiguously represents the data you are reading.

5. In an "aggregate data" section of your script, start with your data frame and code a piped workflow to aggregate/summarize your data.

   Pro Tip: Create your summarized variable names that reflect the metric computed. For example, if you are summarizing the mean of `price` by `carat`, your aggregated data frame should contain the variables `mean_price` and `carat`. Renaming variables (e.g., removing `"mean_"`) later in a plotting workflow is easy to do but if you do not record the variable name well in your aggregate file, you may believe the variable represents a different metric than it actually represents. Sure you can check the code to ensure your assumption is accurate but searching through and reviewing code files is actually more work than simply renaming a variable when needed.

6. Test your code and examine your aggregated data to ensure that your aggregation reflects your intention and makes sense. For team projects, you could also ask for a teammate to review your work to identify any glaring issues. Well-documented scripts and code make collaboration easy, even if you are collaborating with your future self. Think offensively, not defensively.

7. Once you have verified your code is accurate, pass the data frame to `saveRDS()` using `{here}` for managing file paths. Although you may be biased to do so, there is no need to assign the data frame to an object and clutter your workspace. Just save the file as part of your piped workflow.

8. Save your script file.

Note: Make sure that your script and data files are saved in the appropriate project directories.

Part 2: Reading Raw Data and Creating a Point Plot

You will use the full raw (non-aggregated) data set along with `{ggplot2}` to create a `geom_point()` with a numeric variable along the y-axis and the variable that you used for grouping in Part 1 along the x-axis.

1. Create an `.R` script file (Recommendation: Start with `starter_script_file.R`).
2. Name the script file appropriately based on the goal for this task. Refer to Part 1 if you don't remember how.
3. In a "library" section of your script, load your libraries.
4. In a "data-reading" section of your script, write code to read `ggplot2_diamonds.Rds`. Assign the data frame an object name that unambiguously represents the data you are reading.
5. In a "plotting" section of your script, start with your data frame and code a piped workflow to create your point plot. Do not worry about dressing up the plot with any special aesthetics.

   Pro Tip: I highly recommend you specify your function parameters when passing arguments to provide clarity to your code. Saving yourself some keystrokes may not be work the hassle. Parameter naming reduces errors, improves error troubleshooting, and strengthens your knowledge of how the function works. Oh, and if developers change the position order of function parameters and you don't include parameter names when passing arguments to them, code that works today may not work following library updates.
6. Once you are sure that your code creates the intended plot, assign the returned plot to an object name.
7. In typically plotting scripts, you would also include a "plot saving" section of your script. Do not worry about saving the plot object as a `.png` file. At this point, your goal is only to create plots.

Pro Tip: One of your team members should take on the role of creating a function to handle saving all of your plots so that they are uniform in size, dimension, dpi, etc. across the project. More on building functions can be found **in the R Refreshers on the course website**. If you struggle, please meet with me and show me your work so I can help you.

8. Save your plot script file.

Part 3: Creating a Multiple-Layer Point Plot

You will now create a point plot containing *two* layers. Layers of plots can represent different plot types (e.g., points, lines, bars) but they can also represent different data. One layer of your will utilize the raw data set and another layer will utilize the aggregated data that you saved in Part 1. Both layers will use the same x and y variables.

1. Create an `.R` script file named appropriately based on the goal of this part of the exercise. Consider using your previous plot script to create this new script as some elements will be the same (e.g., library, data, first layer of the plot, etc.).

2. In a "library" section of your script, load your libraries.

3. For projects that involve any workflow updating, a "preprocessing" section of your script is really helpful. In such a section, you would `source()` any scripts needed for you to perform your task. Creating a plot with a raw data layer and an aggregated layer requires using data frames. Aggregated data, however, may depend on new cases. For example, many data may change daily, monthly, quarterly, or annually. When your data require aggregation, you want to ensure that you read the most recent aggregated data. The best way to do that is to call/run the script that performs the aggregation and creates the aggregated data file.

   In this section, `source()` any scripts needed to ensure your plot will be up-to-date if a data file were updates.

   Pro Tip: If your team creates a recommended plot saving function script and a plot theme function script, sourcing them is typically not done in this section of a script. Function files are typically best loaded along with libraries which are collections of functions. You would, however, follow the same approach to source scripts in that section to ensure those elements of your plot as well.

4. In a "data-reading" section of your script, write code a) to read the raw data and assign it to an object and b) to read the aggregated data and assign it to an object.

5. In a "plotting" section of your script, start with your raw data frame and code a piped workflow to create your point plot.

6. Add code to create your point plot layer using your aggregated data. Modify your geom color aesthetic so that your point color in the second layer is something other than black or you may not see those against against the raw data.

   Pro Tip: Ensure that you you pass an argument to the `data` parameter of your function. If you don't do this, the geom will inherit the data passed to the `ggplot()` initialization. When data are piped, `ggplot()` inherits the the piped data frame. When data are not piped, `ggplot()` inherits the data frame passed as an argument directly to its `data` parameter.

7. Once you are sure that your code creates a plot with two layers based on two data frames, assign the returned plot to an object name.

8. Save your plot script file.

Project-Related Tip: When preparing your project report, you likely do not want to re-aggregate data within each plot script. This approach would be computationally taxing when dealing with large data files and multiple plots based on the *same* aggregated data. Rather, you would want to ensure that your aggregation scripts all precede any sourcing of *any* aggregated data files or creation of *any* plots. Therefore, the step to `source()` the aggregation script can be commented out of a plot script *if a and only if* you you incorporate those steps in an earlier part of your workflow. This task is something a coding co-lead should handle.

Part 4: Practicing with Git

1. *Stage* your File Change

- Stage your `.R` script file created in Part 1.
- Use `git add <path to file>`

2. *Commit* the File Change

- Commit that change with a clear message.
- Use `git commit -m "added script for ...."`

3. Repeat the staging and committing steps for the other `.R` scripts you created.

4. *Push* Changes

- Push your local repository change to the remote repository on GitHub.
- Use `git push origin main` (WARNING: Do not use `main` for your team project)

5. Check your GitHub Repo for your Changes to `main`