

CHAPTER 2. SOFTWARE AND SCRIPT FILE CONSIDERATIONS

Software Considerations

Avoid Spreadsheet Software and the General User Interface

Over one-third of genomics research includes errors introduced by spreadsheet software (Abeysooriya, Soria, Kasu, & Ziemann, 2021; Lewis, 2021; Ziemann, Eren, & El-Osta, 2016). Specifically, many genomics researchers use spreadsheet-based software (e.g., Microsoft Excel, Google Sheets, LibreOffice) to enter or store data. Unfortunately, however, spreadsheet software's auto-correct feature may misread gene names. For instance, a gene like SEPT2 (Septin-2) may automatically be changed to a date format, such as 2-Sep or 2006-09-02 (Ziemann et al., 2016). As a result, persistent errors plague genomics research, leading to a discipline-wide effort to rename genes in a way that avoids naming conventions that may be misinterpreted and auto-corrected by spreadsheet software.

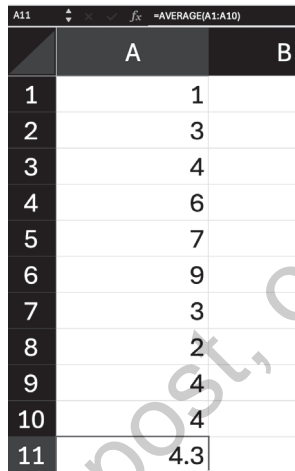
This example is far from the only error that has occurred due to the use of spreadsheet software for data preparation or analysis (for another prominent example, see Chapter 5). Spreadsheet-based software is prone to errors that are easy to make but difficult to detect or trace (Panko, 1998). Said otherwise, the use of spreadsheet-based software hinders transparency. As a result, spreadsheet software is inappropriate for serious data management/analysis, and researchers should avoid using it for *any* data-related task—even simple ones—like renaming variables. In what follows, I describe the advantages of statistical software over spreadsheet software.

Statistical Software Versus Spreadsheet Software

In spreadsheet software, to make changes to data or calculate analyses, researchers typically click on icons or write functions in “cells.” These cells are liable to shift and change, making it easy to accidentally select or edit the wrong cell. In contrast, when using statistical software, researchers write instructions (code) in one file (known as a script file) to tell the software what to do with data that exist in a separate file (a data file). Compared to spreadsheet software, this method is less error-prone and makes it easier to identify issues when they do occur.

So, for example, if I were using a spreadsheet software to calculate the mean of 10 observations in column A (as in Figure 2.1), I might type into cell A11 in the same spreadsheet `= AVERAGE (A1 :A10)`, and it would give me the mean in cell A11.

Figure 2.1 Example of Calculating a Mean in Spreadsheet Software



The image shows a spreadsheet interface. At the top, a formula bar displays `=AVERAGE(A1:A10)`. The spreadsheet has two columns, A and B, and 11 rows. Column A contains the values 1, 3, 4, 6, 7, 9, 3, 2, 4, 4, and 4.3. Column B is empty. The formula bar is at the top, and the spreadsheet grid is below it.

	A	B
1	1	
2	3	
3	4	
4	6	
5	7	
6	9	
7	3	
8	2	
9	4	
10	4	
11	4.3	

Conversely, in a statistical software, I would type code into a separate document, known as a script file. For example, in R, I would type

```
dc_intro_excel %>%
  summarise(mean(A))
```

into the script file, which is in the upper left-hand corner of the R studio interface (see Figure 2.2). This code means, use the `dc_intro_excel` data-base and calculate the mean of column A. Notice that I wrote this code in a file that is distinct from where the data are stored (which is in the “environment,” top right quadrant) or where the output is produced (which is in the “console,” bottom left).

Stata, a different statistical software, operates similarly (see Figure 2.3). In a Stata script file, known as a `.do` file (on the left), I would first load the data and then write the code `sum A` to get the mean of column A in the “results” window (on the right).

Although the software programs are different, unlike a spreadsheet software, neither one involves writing code directly into the data file. All edits are made by writing and executing code in a separate script file.

Figure 2.2 Example of Calculating a Mean in R, a Script File–Based Software

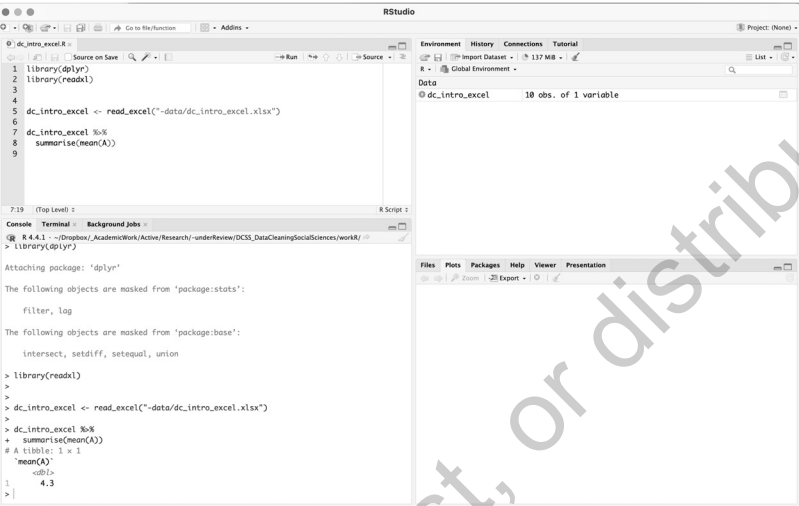
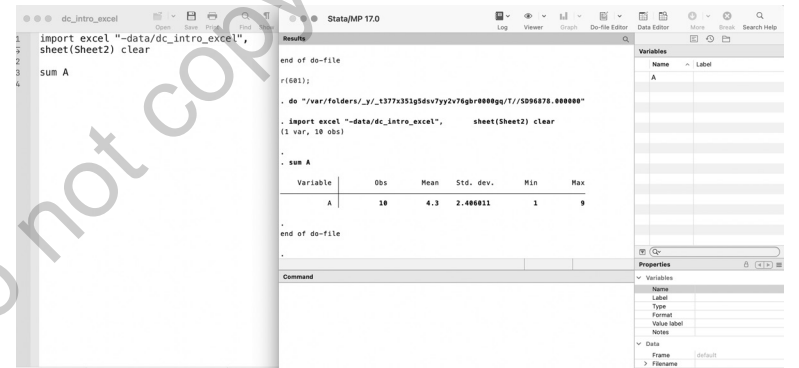


Figure 2.3 Example of Calculating a Mean in Stata, a Script File–Based Software



Additionally, unlike spreadsheet software, the output is produced in a separate window (console for R and review window for Stata) instead of the data file. These features help avoid errors and facilitate accurate, transparent, and efficient data preparation.

Avoid Point-and-Click Methods

While some spreadsheet programs offer features to automate or record actions, they remain heavily dependent on “point-and-click” methods. In general, researchers should avoid using the general user interface (GUI) or point-and-click methods, favoring instead the use of script files (described in the previous section) to execute code. This approach allows for more precise tracking of data management decisions and makes research workflows easier to reproduce and verify.

A simple example of the shortcomings associated with the point-and-click method is to think of detailing the exact steps necessary to make a peanut butter and jelly sandwich. To provide exact details, the researcher would need to list the following:

1. Go to the first cabinet on your left
2. Open the cabinet doors
3. Remove a medium-sized plate
4. Place plate on counter
5. Close cabinet doors
6. Go to second drawer on right
7. Open drawer
8. Remove butter knife
9. Place knife on counter
10. Close drawer
11. Go to cabinet
12. Open cabinet door
13. Remove the bread from the top shelf
14. Close cabinet door
15. Go to the counter where knife and plate reside . . .

I have already listed 15 steps and we haven’t even spread the peanut butter yet! The list of steps quickly becomes tedious and error-prone. In contrast, by using code-based software and script files, researchers can automatically record and reproduce steps. Thus, it is helpful to think of code as the only “real” thing about a project. With code, data preparation

and analysis decisions are automatically recorded.¹ Since these decisions are recorded, they can be easily shared with others, thereby also making them more transparent.

Understand Your Statistical Software

Outside of spreadsheet software, each statistical software package (e.g., R, Stata, SAS, etc.) has its idiosyncrasies that can affect data cleaning processes. These idiosyncrasies include things such as how missing data are handled, how datasets are merged, and how levels of measurement (e.g., continuous or categorical variables) are indicated, among others.

Here are two examples:

1. In Stata, missing data are treated as positive infinity. Therefore, researchers must take care when using arguments that involve greater than symbols/values.
2. In R, when merging two datasets together, researchers must specify that they want to keep observations with missing data. Otherwise, observations with incomplete data may not be included in the merged data.

These examples illustrate how different software programs can affect how researchers approach data cleaning. To ensure efficiency and accuracy of data preparation, researchers must familiarize themselves with the peculiarities of their chosen software and consider how certain decisions may impact their findings. While this book does not delve into the details of software, there are excellent resources for researchers to learn about the way different software “thinks.”² Additionally, since it is difficult to foresee all possible software idiosyncrasies, always verify that the software is doing what you want or expect using code review (discussed in greater detail in Chapter 10).

¹For scholars who do not have a background with coding, it may be difficult to learn new software and how to write code. If necessary, researchers can use point-and-click methods that generate code, which can then be copied and pasted into a script file for rerunning and easier interpretation. This way, the code still exists and facilitates reproducibility. If the point-and-click methods do not generate code, researchers can search for example code, save the code into a script file, and edit it accordingly.

²To accelerate the learning process, I highly recommend workshops. There are also several helpful books and online resources, and I only name a few. For Stata: Acock’s (2010) “A Gentle Introduction to Stata” and Baum’s (2009) “An Introduction to Stata Programming.” For R: <https://cran.r-project.org/doc/manuals/r-release/R-intro.pdf>.

Script File Robustness and Legibility

As noted above, to ensure transparency of research decisions, it is essential to write code into script files. The following sections provide general recommendations for writing script files that can be run and understood by other researchers (or even yourself at a later date—trust me, it can be easy to forget). Examples of script files can be found in the online appendix.

Robustness

Robustness means a script file produces consistent results, regardless of the machine on which it is executed. To achieve robustness, a file must consider all dependencies. As described above, dependencies are packages that need to be downloaded, files that need to be uploaded, and/or specific sections of code that need to be executed before other lines of code can be successfully run. The following section describes ways to reduce file dependencies, thereby ensuring robustness. For templates in Stata and R, see the online supplement.

Start and End with a Clean Slate

Each time researchers open a software program, they should begin and finish with a clean slate (Wickham, Çetinkaya-Rundel, and Grolemund 2023). By clean slate, I am suggesting that at the end of each working session, researchers completely close the program and save all files. Then, at the beginning of a new working session, when researchers open the program, no files automatically open (e.g., data files, script files, project files). By adhering to this practice, researchers develop a habit of saving the necessary code before closing each working session. Additionally, to ensure that any project-specific settings do not accidentally affect all projects, researchers should adopt practices to clear or reset settings at the beginning of each working session.

Exclude File Paths

Since every computer has a different directory (aka file) structure, researchers should exclude file paths (and other hard-coded paths) from script files (Wickham, Çetinkaya-Rundel, and Grolemund 2023; Long, 2009; Taschuk & Wilson, 2017). Even if a project folder has a consistent structure, the path to get to that folder will differ on different computers. Therefore, any code that references a complete file path will not run on another computer. Furthermore, if a project folder is moved or renamed, a researcher would have to change every file path in each and every script

file. To avoid these issues, researchers should set the directory path only once—either manually or in a separate file—distinct from the files used to reproduce analyses.

Load the Data in the Script File

Loading data in the script file prevents researchers from accidentally analyzing the wrong dataset. When a researcher opens a file through the software's GUI, it is difficult for colleagues or other researchers to know which dataset was selected. This lack of transparency can create issues if the wrong dataset was used by mistake or if discrepancies arise in future analyses. By including the code to load the dataset in the script file, researchers can clearly see which data are being used and where they are being used. In other words, loading data in the script automates the importation process, facilitates reproducibility, and minimizes the risk of errors (Wilson et al. 2014).

Include Seeds for Random Numbers

A seed number is the starting point used in the generation of a sequence of random numbers. There are many reasons why a statistical software may use a random-number generator. At times, a researcher may not even realize that a random-number generator is being used for a portion of code.

For example, in multiple imputation, software programs introduce “random error,” meaning that each imputed value varies across observations and datasets. Although the imputed numbers vary, the algorithm used to create the random sequence of numbers is constant. With a seed, the algorithm begins at the same value or random number, creating variation in the same way each time. Without a seed, the random number that it begins at will vary, and the way that the variation is added to the dataset will also vary.

If researchers do not set a seed, then the random number is different each time the file is run. Consequently, the imputation results may vary, and the research findings may also vary. In contrast, if researchers set a seed, they obtain the same sequence of pseudo-random numbers and consistent results. Thus, using seeds promotes reproducibility in random-number generation processes (Long, 2009).

Use Version Control

Over time, the software used for data management and analysis undergoes changes and updates, which can impact reproducibility (Long, 2009; Taschuk & Wilson, 2017). To maintain reproducibility and ensure consistent results, researchers can use version control. In general, version control

is implemented when a researcher can run the same version of software and packages even after a long period of time and after software updates. Version control facilitates collaboration because all collaborators are working with the same software version, regardless of their individual machines.

Depending on the statistical software of interest, version control is implemented differently. In the online supplement, I provide some code that assists with version control. While version control can be accomplished with one or two lines of code for closed-source software programs (e.g., Stata), it requires more effort in open-source software programs (e.g., R). To achieve comprehensive version control in open-source software (such as R), researchers essentially need to create an archive of their entire computer system.

Use Automation

Data cleaning and analysis involve repeating the same processes time and time again. Automation is the process of reducing the number of clicks or lines of code needed to reproduce data cleaning/analysis processes. Reducing lines of code is important since each line of code introduces a potential for error, with estimates ranging between 10 to 25 errors per 1,000 lines of code (McConnell, 2004).³ Therefore, by reducing manual intervention and the number of lines of code, automation reduces the chance of error.

In contrast to automation, hard-coding or the use of “magic numbers” involves directly typing specific values into code that may change over time. Hard-coding is problematic because if those values change based on earlier steps, a researcher would need to manually update every instance where those values are used.

To illustrate, let’s consider an example in which researchers are creating an indicator variable for observations in which the occupational prestige score is an outlier.⁴

Although there are many ways to calculate outliers, one definition is any value greater than or less than the mean, plus or minus 2.24 standard deviations ($\mu \pm (2.24 \times \delta)$).⁵ For instance, the mean occupational prestige score in the General Social Survey as of 2018 is 44.68 and the standard deviation

³ GUI and point-and-click methods are not any less error-prone. In contrast, they may introduce even more errors and reduce reproducibility.

⁴ An outlier is a value that is markedly different from others. I discuss outliers in detail in Chapter 7. For now, just think of it as trying to identify strange values.

⁵ This is the equation for a confidence interval, where μ refers to the mean (pronounced like “myoo”), and δ refers to the standard deviation. For more information on means and standard deviations, see Chapter 7.

is 13.65. This would mean that the upper limit is 75.26,⁶ and any value greater than 75.26 would be an outlier.

In the manual, “magic number,” or hard-coding approach, researchers could generate a new variable equal to 1 or an “Outlier” if the observation’s value of the variable is greater than the upper limit and otherwise equal to 0 or “Not Outlier.” The code for this would look like

- Calculate the mean (44.68)
- Calculate the standard deviation (13.65)
- Calculate the mean plus 2.24 standard deviations (75.26)
- Create a new variable called `upper_outlier`
- Set `upper_outlier = 1` if `prestige` is greater than 75.26
- Set `upper_outlier = 0` if `prestige` is not greater than 75.26

Unfortunately, with this approach, if the mean or standard deviation ever changes—for example, if researchers drop or add observations—they would need to go back and find these hard-coded values and edit them.

Alternatively, in the automated approach, researchers avoid typing the mean and standard deviation in the code. Rather, the researchers would create a local, named constant, or temporary object to store these values. For example, rather than type the mean and standard deviation manually, they could use code that says

- Calculate the mean
- Save this value to “mean”, so that when I say “mean”, the calculated mean appears in its place
- Calculate the standard deviation
- Save this value to “sd” so that when I say “sd”, the calculated standard deviation appears in its place
- Create a new variable called `upper_outlier`
- Set `upper_outlier = 1` if `prestige` is greater than “mean” + (“sd” * 2.24)
- Set `upper_outlier = 0` if `prestige` is not greater than “mean” + (“sd” * 2.24)

⁶44.68 + (2.24 * 13.65)

By using automation, researchers ensure that changes in the data cleaning process, such as removing observations with missing data, will automatically update the relevant values.

One can see how, throughout the process of data cleaning and modeling, there is a great potential for “hard-coding.” Unfortunately, a hard-coded value requires a researcher to identify and modify it each time the data change. Instead, by using automation, researchers can reduce the number of things they are required to remember such as the location of these values. Furthermore, automation eliminates concerns about accurately typing each number, as the numbers are automatically incorporated into the code.

Automation also aids in error detection. When using automation, a single error results in consistent, systematic errors, which may be easier to detect than a single, non-systematic error (Long, 2009; Wilson et al., 2014). Additionally, automation makes it faster to respond to errors or changes. Rather than make numerous small edits, a researcher can make one edit that will carry throughout the project. In summary, by using automation, researchers reduce the potential points of origin for mistakes and are better able to catch mistakes when they do occur (Yarkoni et al., 2019).

Legibility

Legibility refers to how easy or difficult it is to comprehend the code from a project. Achieving legibility requires the use and implementation of the following principles.

Internal Documentation

To make it easier to navigate code, it is helpful to include explanatory notes to oneself, coauthors, and interested readers. Although good labels and names reduce the need for extensive notes (as discussed in more detail in Chapter 3), internal documentation within files remains important.

A key element of internal documentation is provenance, which should exist in all figures and tables (Buneman, Chapman, & Cheney, 2006; Long, 2009). Provenance is a small piece of text located in an inconspicuous place that usually includes the name of the script file used to generate the table or figure. Additionally, all script files should have their own provenance, usually listed at the top of a file (Long, 2009). This section of the script file should include the following information:

- Author name
- Name of person who most recently edited the file

- Date created
- Date of last update
- File name

For an example of how to implement internal documentation in code files, see the template script files in Stata and R provided in the online appendix.

Alignment, Indentation, and Attention

Consistent alignment and indentation make code more visually organized. When code is thoughtfully spaced, it becomes easier to read and edit. That is because alignment and indentation can be used to emphasize commands and highlight variables.

For instance, consider the following code:

```
reg yvar xvar1 xvar2 xvar3 xvar4
      xvar5 xvar6
```

In this example, the alignment and indentation highlight the command “regression” and the dependent variable “yvar.” On the other hand, the following code appears more cluttered, making it harder to distinguish the command and dependent variable:

```
reg yvar xvar1 xvar2 xvar3 xvar4
xvar5 xvar6
```

In addition to spacing and alignment, largely aesthetic choices also impact code legibility and should be consistent within projects. Although specific decisions may vary within teams or by software, there are some commonly agreed-upon best practices (Wickham, 2019). The following bullet points provide some guidance:

- Spaces and Tabs
 - For indentation, use spaces instead of tabs, as tabs can have varying formatting on different machines.
 - Avoid mixing spaces and tabs.
- Attention
 - To divide files into easily readable chunks, use symbols that are not processed by the software, but allow for comments or draw attention to specific areas (e.g., in Stata: // or *; in R: # or -).

- Line Length
 - Keep lines of code short, preferably under 80 characters.⁷
 - To keep lines of code short, consider using locals or vectors of variable names instead of repeatedly rewriting long lists of variables.
 - When possible, consider using abbreviated commands or functions.

Summary

In summary, for transparency and accuracy, researchers should conduct their data preparation and analyses in a code-based statistical software (as opposed to a spreadsheet software), using code (as opposed to a general user interface or GUI) that is saved to script files. Transparency is further facilitated by the public sharing of data and script files (Leek & Peng, 2015). To optimize the transparency of script files, they should be robust (can run on any machine and produce identical results) and legible (easy to read) (Wilson et al., 2014). To ensure robust script files, researchers should exclude file paths, load data within the script files, use seeds for random numbers, adopt version control, and implement automation to reduce errors and the need for manual intervention. To ensure legible script files, researchers should include provenance in all files and adopt thoughtful and consistent practices for aesthetic choices like spacing and drawing attention to sections of code.

⁷This advice originated from the practice of occasionally printing code on paper, which is now uncommon. Despite the low likelihood of printing code, shorter lines are still easier to read and debug on computer screens, and 80 characters is a helpful guideline.