Proctor Foundation Data Science Handbook

Contributors: Ben Arnold, Jade Benjamin-Chung, Kunal Mishra, Anna Nguyen, Nolan Pokpongkiat, Stephanie Djajadi, Eric Kim (many from UC Berkeley in addition to Proctor)

2019-12-06

Contents

Welcome!

Welcome to the Francis I. Proctor Foundation at the University of California, San Francisco (https://proctor.ucsf.edu)!

This handbook summarizes some best practices for data science, drawing from our experience at the Francis I. Proctor Foundation and from that of our close colleagues in the Division of Epidemiology and Biostatistics at the University of California, Berkeley (where Prof. Ben Arnold worked for many years before joining Proctor).

We do not intend this handbook to be a comprehensive guide to data science. Instead, it focuses more on practical, "how-to" guidance for conducting data science within epidemiologic research studies. Where possible, we reference existing materials and guides.

Although many of the ideas of environment-independent, the examples draw from the R programming language. For an excellent overview of data science in R, see the book R for Data Science.

Much of the material in this handbook evolved from a version of Dr. Jade Benjamin-Chung's lab manual at the University of California, Berkeley. In addition to the Proctor team, many contributors include current and former students from UC Berkeley.

The last two chapters of the handbook cover our communication strategy and code of conduct for team members who work with Prof. Ben Arnold, who leads Proctor's Data Coordinating Center. They summarize key pieces of a functional data science team. Although the last two chapters might be of interest to a broader circle, *they are mostly relevant for people working directly with Ben.* Just because they are at the end does not make them less important.

It is a living document that we strive to update regularly. If you would like to contribute, please write Ben (ben.arnold@ucsf.edu) and/or submit a pull request.

The GitHub repository for this handbook is: https://github.com/proctor-ucsf/dcc-handbook

Chapter 1

Introduction: Work Flow and Reproducible Analyses

Contributors: Ben Arnold

This handbook collates a number of tips to help organize the workflow of epidemiologic data analyses. There are probably a dozen good ways to organize a workflow for reproducible research. This document includes recommendations that arise from our own team's experience through numerous field trials and observational data analyses. The recommendations will not work for everybody or for all applications. But, they work well for most of us most of the time, else we wouldn't put in the time to share them.

Start with two organizing concepts:

- **Workflow**. Defined here as the process required to draw scientific inference from data collected in the field or lab. I.e., the process by which we take data, and then process it, share it internally, analyze it, and communicate results to the scientific community.

- **Reproducible research**. A fundamental characteristic of the scientific method is that study findings can be reproduced beyond the original investigators. Data analyses that contribute to scientific research should be described and organized in a way that they could be reproduced by an independent person or research group. A data analysis that is not reproducible violates a core principle of the scientific method.

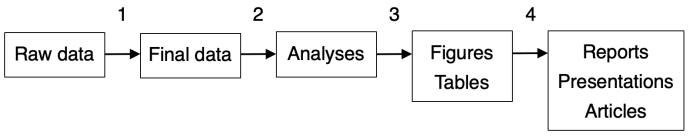


Figure 1.1: Overview of the four main steps in a typical data science workflow

1.1 Workflow

Broadly speaking, a typical scientific data science work flow involves four steps to transform raw data (e.g., from the field) into summaries that communicate results to the scientific community.

When starting a new project, the work flow tends to evolve gradually and by iteration. Data cleaning, data processing, exploratory analyses, back to data cleaning, and so forth. If the work takes place in an unstructured environment with no system to organize files and work flow, it rapidly devolves into into a disorganized mess; analyses become difficult or impossible to replicate and they are anything but scientific. Projects with short deadlines (e.g., proposals, conference abstract submissions, article revisions) are particularly vulnerable to this type of organizational entropy. Putting together a directory and workflow plan from the start helps keep files organized and prevent disorder. Modifications are inevitable – as long as the system is organized, modifications are usually no problem.

Depending on the project, each step involves a different amount of work. Step 1 is by far the most time consuming, and often the most error-prone. We devote an entire chapter to it below (Data cleaning and processing)

1.2 Reproducibility

As a guiding directive, this process should be reproducible. If you are not familiar with the concept of reproducible research, start with this manifesto (Munafo et al. 2017). For a deeper dive, we highly recommend the recent book from Christensen, Freese, and Miguel (2019). Although it is framed around social science, the ideas apply generally.

1.3 Automation

We recommend that the workflow be as automated as possible using a programming language. Automating the workflow in a programming language, and essentially reducing it to text, is advantageous because it makes the process

transparent, well documented, easily modified, and amenable to version control; these characteristics lend themselves to reproducible research.

At Proctor, we mostly use R. With the development of Rstudio, R Markdown and the tidyverse ecosystem (among others), the R language has evolved as much in the past few years as in all previous decades since its inception. This has made the conduct of automated, reproducible research considerably easier than it was 10 years ago.

If you have a step in your analysis workflow that involves point-and-click or copy/paste, then STOP, and ask yourself (and your team): *How can I automate this?*

Chapter 2

Workflows

Contributors: Ben Arnold

A data science work flow typically progresses through 4 steps that rarely evolve in a purely linear fashion, but in the end should flow in this direction:

Table 2.1: Workflow basics

Steps	Example activities	⇒ Inputs	⇒ Outputs
1	**Data cleaning and processing**		
.	make a plan for final datasets, fix data entry errors, create derived variables, plan for public replication files	untouched datasets	final datasets
2-3	**Analyses**		

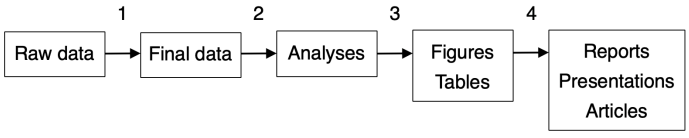


Figure 2.1: Overview of the four main steps in a typical data science workflow

Steps	Example activities	⇒ Inputs	⇒ Outputs
.	exploratory data analysis, study monitoring, summary statistics, statistical analyses, independent replication of analyses, make figures and tables	final datasets	saved results (.rds/.csv), tables (.html,.pdf), figures (.html/.png)
4 .	**Communication** results synthesis	saved results, figures, tables	monitoring reports, presentations, scientific articles

In many modern data science workflows, steps 2-4 can be accomplished in a single R notebook or Jupyter notebook: the statistical analysis, creation of figures and tables, and creation of reports.

However, it is still useful to think of the distinct stages in many cases. For example, a single statistical analysis might contribute to a DSMC report, a scientific conference presentation, and a scientific article. In this example, each piece of scientific communication would take the same input (stored analysis results as .csv/.rds) and then proceed along slightly different downstream workflows.

It would be more error prone to replicate the same statistical analysis in three parallel downstream work flows. This illustrates a key idea that holds more generally:

Key idea for workflows: Whenever possible, avoid repeating the same data processing or statistical a

Chapter 3

Directory Structure and Code Repositories

Contributors: Kunal Mishra, Jade Benjamin-Chung, Ben Arnold

The backbone of your project workflow is the file directory so it makes sense to spend time organizing the directory. Note that `directory` is the technical term for the system used to organize individual files. Most non-UNIX environments use a folder analogy, and directory and folder can be used interchangeably in a lot of cases. A well organized directory will make everything that follows much easier. Just like a well designed kitchen is essential to enjoy cooking (and avoid clutter), a well designed directory helps you enjoy working and stay organized in a complex project with literally thousands of related files. Just like a disorganized kitchen ("now where did I put that spatula?") a disorganized project directory creates confusion, lost time, stress, and mistakes.

Another huge advantage of maintaining a regular/predictable directory structure within a project and across projects is that it makes it more intuitive. When a directory is intuitive, it is easier to work collaboratively across a larger team; everybody can predict (at least approximately) where files should be.

Nested within your directory will be a `code repository`. Sometimes we find it useful to manage the code repository using version control, such as git/GitHub.

Other chapters will discuss coding practices, data management, and GitHub/version control that will build from the material here.

Carrying the kitchen analogy further: here, we are designing the kitchen. Then, we'll discuss approaches for how to cook in the kitchen that we designed/built.

3.1 Small and large projects

Our experience is that the overwhelming majority of projects come in two sizes: small and large. We recommend setting up your directory structure depending on how large you expect the project to be. Sometimes, small projects evolve into large projects, but only occasionally. A small project is something like a single data analysis with a single published article in mind. A large project is an epidemiologic field study, where there are multiple different types of data and different types of analyses (e.g., sample size calculations, survey data, biospecimens, substudies, etc.).

Small project: There is essentially one dataset and a single, coherent analysis. For example, a simulation study or a methodology study that will lead to a single article.

Large project: A field study that includes multiple activities, each of which generates data files. Multiple analyses are envisioned, leading to multiple scientific articles.

Large projects are more common and more complicated. Most of this chapter focuses on large project organization (small projects can be thought of as essentially one piece of a large project).

3.2 Directory Structure

In the example below, we follow a basic directory naming convention that makes working in UNIX and typing directory statements in programs much easier:

- **short names**
- **no spaces in the names** (not essential but a personal preference. Can use `_` or `-` instead)
- **lower case** (not essential, again, personal preferences vary!)

For example, Ben completed a study in Tamil Nadu, India during his dissertation to study the effect of improvements in water supply and sanitation on child health. Instead of naming the directory `Tamil Nadu` or `Tamil Nadu WASH Study`, he used `trichy` instead (a colloquial name for the city near the study, Tiruchirappalli), which was much easier to type in the terminal and in directory statements. A short name helps make directory references easier while programming.

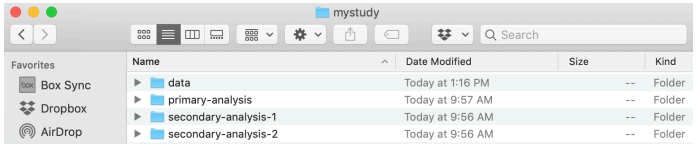


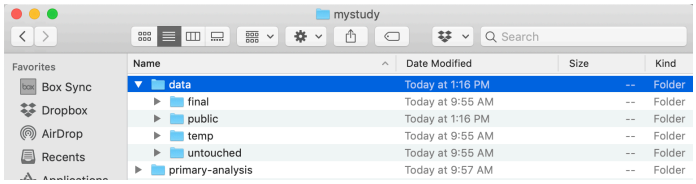
Figure 3.1: Example directory for 'mystudy'

3.2.1 First level: data and analyses

Start by dividing a project into major activities. In the example abpve, the project is named `mystudy`. There is a `data` subdirectory (more in a sec), and then three major activities, each corresponding to a separate analysis: `primary-analysis`,`secondary-analysis-1`, and `secondary-analysis-2`. In a real project, the names could be more informative, such as "trachoma-qpcr". Also, a real project might also include many additional subdirectories related to administrative and logistics activities that do not relate to data science, such as irb, travel, contracts, budget, survey forms, etc.).

Dividing files into major activities helps keep things organized for really big projects. In a multi-site study, consider including a directory for each site before splitting files into major activities. Ideally, analyses will not span major activity subdirectories in a project folder, but sometimes you can't predict/avoid that from happening.

3.2.2 Second level: data



Each project will include a `data` directory. We recommend organizing it into 3 parts: `untouched`, `temp`, and `final`. Often, it is useful to include a fourth subdirectory called `public` for sharing public versions of datasets.

The `untouched` directory includes all untouched datasets that are used for the study. Once saved in the directory never touch them; you will read them into the work flow, but **never, ever save over them**. If the study has repeated extracts from rolling data collection or electronic health records, consider subdirectories within `untouched` that are indexed by date.

The `temp` directory (optional, not essential) includes temporary files that you might generate during the data management process. This is mainly a space for

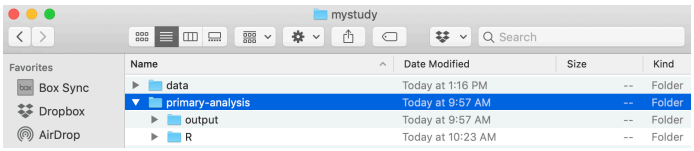
experimentation. As a rule, never save anything in the temp directory that you cannot delete. Regularly delete files in the temp directory to save disk space.

The `final` directory includes final datasets for the activity. Final datasets are de-identified and require no further processing; they are clean and ready for analysis. They should be accompanied by meta-data, which at minimum includes the data's provenance (i.e., how it was created) and what it includes (i.e., level of the data, plus variable coding/labels/descriptions). Clean/final datasets generated by one analysis might be reused in another.

3.2.3 Second level: analysis

We recommend maintaining a separate subdirectory for each major analysis in a project. In this example, there are three with not-very-creative names from the view of trial: `primary-analysis`, `secondary-analysis-1`, `secondary-analysis-2`.

Think of each analysis as the scope of all of the work for a single, published paper. We recommend dividing the analysis project into a space for computational notebooks / scripts (i.e., a `code repository`), and a second for their output. The reason for the split is to make it easier to use version control (should you choose) for the code. Version control like `git` and `GitHub` (see the Chapter on GitHub) works well for text files but isn't really designed for binary files such as images (.png), datasets (.rds), or PDF files (.pdf). It is certainly possible to use git with those file types, but since git makes a new copy of the file every time it is changed the git repo can get horribly bloated and takes up too much space on disk. Consolidating the output into a separate directory makes it more obvious that it isn't under version control. In this example, there are separate parts for code (`R`) and output (`output`). Output could include figures, tables, or saved analysis results stored as data files (.rds or .csv). Another conventional name for the code repository is `src` as an alternative to `R` if you use other languages.

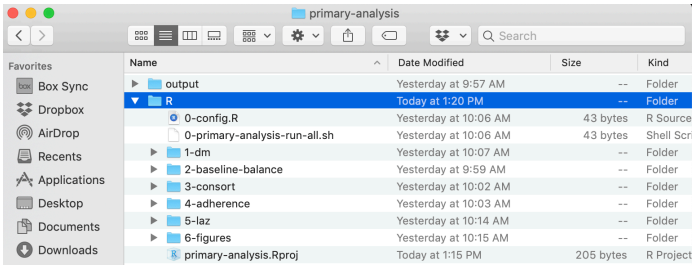


Interdependence between analyses: Sometimes a result from an analysis might be a cleaned dataset that could feed into future, distinct analyses. This is quite common, for example, in large trials where a set of baseline characteristics might be used in multiple separate papers for different endpoints, either for assessing balance of the trial population or subgroups, or used as adjustment covariates in additional analyses of the trial. In this case, the cleaned dataset would be written to the `data/final` directory and is thus available for future use.

3.3 Code Repositories

Maintain a separate code repository for each major analysis activity (last section).

We recommend the following structure for a code repository:



With subdirectories that generally look like this:

```
.gitignore
primary-analysis.Rproj
0-config.R
0-shared-functions.R
0-primary-analysis-run-all.sh
1-dm /
    0-dm-run-all.sh
    1-format-enrollment-data.R
    2-format-adherence-data.R
    3-format-LAZ-measurements.R
2-baseline-balance /
    0-baseline-balance-run-all.sh
    ...
3-consort /
    0-consort-run-all.sh
    ...
4-adherence /
    0-adherence-run-all.sh
    ...
5-laz /
    0-laz-run-all.sh
    1-laz-unadjusted-analysis.R
    2-laz-adjusted-analysis.R
6-figures /
    0-figures-run-all.sh
    Fig1-consort.Rmd
```

```
Fig2-adherence.Rmd
Fig3-laz.Rmd
```

Note `dm` is shorthand for "data management." You can call the data management directory anything you want, but just ensure that you have one. This helps ensure work conducted in step 1 of your workflow stays upstream from all analyses (see Chapter on workflows). Also note that in this example, all of the scripts are `.R` files. Increasingly, we use R Markdown notebooks `.Rmd` instead of / in addition to R files.

For brevity, we haven't expanded every directory, but you can glean some important takeaways from what you *do* see.

3.3.1 `.Rproj` files

An "R Project" can be created within RStudio by going to `File >> New Project`. Depending on where you are with your research, choose the most appropriate option. This will save preferences, working directories, and even the results of running code/data (though we recommend starting from scratch each time you open your project, in general). Then, ensure that whenever you are working on that specific research project, you open your created project to enable the full utility of `.Rproj` files. This also automatically sets the directory to the top level of the project.

3.3.2 Configuration ('config') File

This is the single most important file for your project. It will be responsible for a variety of common tasks, declare global variables, load functions, declare paths, and more. *Every other file in the project* will begin with `source("0-config")`, and its role is to reduce redundancy and create an abstraction layer that allows you to make changes in one place (`0-config.R`) rather than 5 different files. To this end, paths that will be referenced in multiple scripts (e.g., a `clean_data_path`) can be declared in `0-config.R` and simply referred to by its variable name in scripts. If you ever want to change things, rename them, or even switch from a downsample to the full data, all you would then to need to do is modify the path in one place and the change will automatically update throughout your project. See the example config file for more details. The paths defined in the `0-config.R` file assume that users have opened the `.Rproj` file, which sets the directory to the top level of the project.

This GitHub repository that has replication files for this study includes an example of a streamlined `config.R` file, with all packages loaded and directory references defined.

3.3.3 Shared Functions File

If you write a custom function for an analysis and need to use it repeatedly across multiple analysis scripts, then it is better to consolidate it into a single shared functions script and source that file into the analysis scripts. The reason for this is that it enables you to edit the function in a single place and ensure that the changes are implemented across your entire workflow. In extreme cases, you might have so many shared functions that you need an entire subdirectory with separate scripts. This repository includes an example (`0-project-functions`) of a large analysis (currently still a work in progress).

3.3.4 Order Files and Subdirectories

This makes the jumble of alphabetized filenames much more coherent and places similar code and files next to one another. Although sometimes there is not a linear progression from 1 to 2 to 3, in general the structure helps reflect how data flows from start to finish and allows us to easily map a script to its output (i.e. `primary-analysis/R/5-laz/1-laz-unadjusted-analysis.R` => `primary-analysis/output/5-laz/1-laz-unadjusted-analysis.RDS`). That is, the code repository and the output are approximately mirrored. If you take nothing else away from this guide, this is the single most helpful suggestion to make your workflow more coherent. Often the particular order of files will be in flux until an analysis is close to completion. At that time it is important to review file order and naming and reproduce everything prior to drafting a manuscript.

In the `6-figures` subdirectory, each RMarkdown file (computational notebook) is linked to a specific figure in a hypothetical manuscript. This makes it easier to link specific notebooks in figure legends, and to see which file creates each figure.

3.3.5 Use Bash scripts to ensure reproducibility

Bash scripts are useful components of a reproducible workflow. At many of the directory levels (i.e. in `5-laz`), there is a bash script that runs each of the analysis scripts. This is exceptionally useful when data "upstream" changes – you simply run the bash script. See the UNIX Chapter for further details.

3.3.6 Alternative approach for code repos

Another approach for organizing your code repository is to name all of your scripts according to the final figure or table that they generate for a particular article. In our experience, this *only* works for small projects, with a single set of coherent analyses. Here, you might have an alternative structure such as:

```
.gitignore
primary-analysis.Rproj
0-config.R
0-shared-functions.R
0-primary-analysis-run-all.sh
1-dm /
    0-dm-run-all.sh
    1-format-enrollment-data.R
    2-format-adherence-data.R
    3-format-LAZ-measurements.R
Fig1-consort.Rmd
Fig2-adherence.Rmd
Fig3-1-laz-analysis.Rmd
Fig3-2-laz-make-figure.Rmd
```

There is still a need for a separate data management directory (e.g., `dm`) to ensure that workflow is upstream from the analysis (more below in chapter on UNIX), but then scripts are all together with clear labels. If a figure requires two stages to the analysis, then you can name them sequentially, such as `Fig3-1-laz-analysis.Rmd`, `Fig3-2-laz-make-figure.Rmd`. There is no way to divine how all of the analyses will neatly fit into files that correspond to separate figures. Instead, they will converge on these file names through the writing process, often through comsolidation or recombination.

One example of a small repo is here: https://github.com/ben-arnold/enterics-seroepi

Chapter 4

Coding Practices

Contributors: Kunal Mishra, Jade Benjamin-Chung, Ben Arnold

4.1 Organizing scripts

Just as your data "flows" through your project, data should flow naturally through a script. Very generally, you want to:

1. describe the work completed in the script in a comment header
2. source your configuration file (`0-config.R`)
3. load all of your data
4. do all your analysis/computation in order
5. save your results

Each section should be "chunked together" using comments, often with many chunks in a single section. See this file for a good example of how to cleanly organize a file in a way that follows this "flow" and functionally separate pieces of code that are doing different things. This is another example in a `.Rmd` format, where chunking is made even more obvious by the interleaving of markdown text and R code in the same notebook file.

4.2 Documenting your code

4.2.1 File headers

Every file in a project should have a header that allows it to be interpreted on its own. It should include the name of the project and a short description for what this file (among the many in your project) does specifically. You may optionally

wish to include the inputs and outputs of the script as well, though the next section makes this significantly less necessary. It can be very helpful to include your name and email address as well so others can identify who wrote the code. This is unnecessary if you are using version control (`git/GitHub`) becuase that information will be tracked by commits.

```
#------------------------------------------------------------------------------
# @Organization - Example Organization
# @Project - Example Project
# @Author - Your name, and possibly email address (if appropriate)
# @Description - This file is responsible for [...]
#------------------------------------------------------------------------------
```

Consider using RStudio's code folding feature to collapse and expand different sections of your code. Any comment line with at least four trailing dashes (-), equal signs (=), or pound signs (#) automatically creates a code section. Delimiters for chunks are a personal preference and all work equally well. For example:

```
# Section 1 ------------------
```

works equally well as

```
##############################
# Section 1 ##################
##############################
```

4.2.2 Comments in the body of your script

Commenting your code is an important part of reproducibility and helps document your code for the future. When things change or break, you'll be thankful for comments. When you revisit code you wrote two years earlier, you'll be thankful for comments. There's no need to comment excessively or unnecessarily, but a comment describing what a large or complex chunk of code does is always helpful. See this file for an example of how to comment your code and notice that comments are always in the form of:

```
# This is a comment -- first letter is capitalized and spaced
away from the pound sign
```

4.2.3 Function documentation

Every function you write must include a header to document its purpose, inputs, and outputs. For any reproducible workflows, they are essential, because R is dynamically typed. This means, you can pass a `string` into an argument that is meant to be a `data.table`, or a `list` into an argument meant for a `tibble`. It is the responsibility of a function's author to document what each argument is

meant to do and its basic type. This is an example for documenting a function (inspired by JavaDocs, R's Plumber API docs, and Roxygen2):

```
#------------------------------------------------------------------------------
# Documentation: calc_fluseas_mean
# Usage: calc_fluseas_mean(data, yname)
# @description: Make a dataframe with rows for flu season and site
#               and the number of patients with an outcome, the total patients,
#               and the percent of patients with the outcome

# Arguments/Options:
# @param data: a data frame with variables flu_season, site, studyID, and yname
# @param yname: a string for the outcome name
# @param silent: a boolean specifying whether the function shouldn't output anything to the consc

# @return: the dataframe as described above
# @output: prints the data frame described above if silent is not True
#------------------------------------------------------------------------------

calc_fluseas_mean = function(data, yname, silent = TRUE) {
 ### function code here

}
```

The header tells you what the function does, its various inputs, and how you might go about using the function to do what you want. Also notice that all optional arguments (i.e. ones with pre-specified defaults) follow arguments that require user input.

- **Note**: As someone trying to call a function, it is possible to access a function's documentation (and internal code) by `CMD-Left-Click`ing the function's name in RStudio

- **Note**: Depending on how important your function is, the complexity of your function code, and the complexity of different types of data in your project, you can also add "type-checking" to your function with the `assertthat::assert_that()` function. You can, for example, `assert_that(is.data.frame(statistical_input))`, which will ensure that collaborators or reviewers of your project attempting to use your function are using it in the way that it is intended by calling it with (at the minimum) the correct type of arguments. You can extend this to ensure that certain assumptions regarding the inputs are fulfilled as well (i.e. that `time_column`, `location_column`, `value_column`, and `population_column` all exist within the `statistical_input` tibble).

4.3 Object naming

Generally we recommend using nouns for objects and verbs for functions. This is because functions are performing actions, while objects are not.

Try to make your variable names both more expressive and more explicit. Being a bit more verbose is useful and easy in the age of autocompletion! For example, instead of naming a variable `vaxcov_1718`, try naming it `vaccination_coverage_2017_18`. Similarly, `flu_res` could be named `absentee_flu_residuals`, making your code more readable and explicit.

- For more help, check out Be Expressive: How to Give Your Variables Better Names

We recommend you use **Snake_Case**.

- Base R allows `.` in variable names and functions (such as `read.csv()`), but this goes against best practices for variable naming in many other coding languages. For consistency's sake, `snake_case` has been adopted across languages, and modern packages and functions typically use it (i.e. `readr::read_csv()`). As a very general rule of thumb, if a package you're using doesn't use `snake_case`, there may be an updated version or more modern package that *does*, bringing with it the variety of performance improvements and bug fixes inherent in more mature and modern software.
- **Note**: you may also see `camelCase` throughout the R code you come across. This is *okay* but not ideal – try to stay consistent across all your code with `snake_case`.
- **Note**: there is nothing inherently wrong with using `.` in variable names, just that it goes against style best practices that are cropping up in data science, so its worth getting rid of these bad habits now.

4.4 Function calls

In a function call, use "named arguments" and separate arguments by to make your code more readable.

Here's an example of what not to do when calling the function a function `calc_fluseas_mean` (defined above):

```
mean_Y = calc_fluseas_mean(flu_data, "maari_yn", FALSE)
```

And here it is again using the best practices we've outlined:

```
mean_Y = calc_fluseas_mean(
  data = flu_data,
  yname = "maari_yn",
  silent = FALSE
```

)

4.5 The here package

The `here` package is one great R package that helps multiple collaborators deal with the mess that is working directories within an R project structure. Let's say we have an R project at the path `/home/oski/Some-R-Project`. My collaborator might clone the repository and work with it at some other path, such as `/home/bear/R-Code/Some-R-Project`. Dealing with working directories and paths explicitly can be a very large pain, and as you might imagine, setting up a Config with paths requires those paths to flexibly work for all contributors to a project. This is where the `here` package comes in and this a great vignette describing it.

For more motivation on why you should use the `here` and R projects (`.Rproj`), read this excellent blog post from Tidyverse.

4.6 Tidyverse

Throughout this document there have been references to the Tidyverse, but this section is to explicitly show you how to transform your Base R tendencies to Tidyverse (or Data.Table, Tidyverse's performance-optimized competitor). For most of our work that does not utilize very large datasets, we recommend that you code in Tidyverse rather than Base R. Tidyverse is quickly becoming the gold standard in R data analysis and modern data science packages and code should use Tidyverse style and packages unless there's a significant reason not to (i.e. big data pipelines that would benefit from Data.Table's performance optimizations).

The package author has published a great textbook on R for Data Science, which leans heavily on many Tidyverse packages and may be worth checking out.

The following list is not exhaustive, but is a compact overview to begin to translate Base R into something better:

Base R	Better Style, Performance, and Utility
`read.csv()`	`readr::read_csv()` or `data.table::fread()`
`write.csv()`	`readr::write_csv()` or `data.table::fwrite()`
`readRDS`	`readr::read_rds()`
`saveRDS()`	`readr::write_rds()`

Base R	Better Style, Performance, and Utility
—	—
`data.frame()`	`tibble::tibble()` or `data.table::data.table()`
`rbind()`	`dplyr::bind_rows()`
`cbind()`	`dplyr::bind_cols()`
`df$some_column`	`df %>% dplyr::pull(some_column)`
`df$some_column = ...`	`df %>% dplyr::mutate(some_column = ...)`
`df[get_rows_condition,]`	`df %>% dplyr::filter(get_rows_condition)`
`df[,c(col1, col2)]`	`df %>% dplyr::select(col1, col2)`
`merge(df1, df2, by = ..., all.x = ..., all.y = ...)`	`df1 %>% dplyr::left_join(df2, by = ...)` or `dplyr::full_join` or `dplyr::inner_join` or `dplyr::right_join`
—	—
`str()`	`dplyr::glimpse()`
`grep(pattern, x)`	`stringr::str_which(string, pattern)`
`gsub(pattern, replacement, x)`	`stringr::str_replace(string, pattern, replacement)`
`ifelse(test_expression, yes, no)`	`if_else(condition, true, false)`
Nested: `ifelse(test_expression1, yes1, ifelse(test_expression2, yes2, ifelse(test_expression3, yes3, no)))`	`case_when(test_expression1 ~ yes1, test_expression2 ~ yes2, test_expression3 ~ yes3, TRUE ~ no)`
`proc.time()`	`tictoc::tic()` and `tictoc::toc()`
`stopifnot()`	`assertthat::assert_that()` or `assertthat::see_if()` or `assertthat::validate_that()`

For a more extensive set of syntactical translations to Tidyverse, you can check out this document.

Working with Tidyverse within functions can be somewhat of a pain due to non-standard evaluation (NSE) semantics. If you're an avid function writer, we'd recommend checking out the following resources:

- Tidy Eval in 5 Minutes (video)
- Tidy Evaluation (e-book)
- Data Frame Columns as Arguments to Dplyr Functions (blog)
- Standard Evaluation for *_join (stackoverflow)
- Programming with dplyr (package vignette)

4.7 Coding with R and Python

If you're using both R and Python, you may wish to check out the Feather package for exchanging data between the two languages extremely quickly.

Chapter 5

Coding Style

Contributors: Kunal Mishra, Jade Benjamin-Chung, and Ben Arnold

5.1 Line breaks

- For `ggplot` calls and `dplyr` pipelines, do not crowd single lines. Here are some nontrivial examples of "beautiful" pipelines, where beauty is defined by coherence:

```
# Example 1
school_names <- list(
  OUSD_school_names = absentee_all %>%
    filter(dist.n == 1) %>%
    pull(school) %>%
    unique() %>%
    sort(),

  WCCSD_school_names <- absentee_all %>%
    filter(dist.n == 0) %>%
    pull(school) %>%
    unique() %>%
    sort()
)
```

```
# Example 2
absentee_all <- fread(file = raw_data_path) %>%
  mutate(program = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                             schoolyr %in% program_schoolyrs ~ 1)) %>%
  mutate(period = case_when(schoolyr %in% pre_program_schoolyrs ~ 0,
                            schoolyr %in% LAIV_schoolyrs ~ 1,
```

```
                                schoolyr %in% IIV_schoolyrs ~ 2)) %>%
  filter(schoolyr != "2017-18")
```

And of a complex `ggplot` call:

```
# Example 3
ggplot(data=data,
       mapping=aes_string(x="year", y="rd", group=group)) +

  geom_point(mapping=aes_string(col=group, shape=group),
             position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping=aes_string(ymin="lb", ymax="ub", col=group),
                position=position_dodge(width=0.2),
                width=0.2) +

  geom_point(position=position_dodge(width=0.2),
             size=2.5) +

  geom_errorbar(mapping=aes(ymin=lb, ymax=ub),
                position=position_dodge(width=0.2),
                width=0.1) +

  scale_y_continuous(limits=limits,
                     breaks=breaks,
                     labels=breaks) +

  scale_color_manual(std_legend_title,values=cols,labels=legend_label) +
  scale_shape_manual(std_legend_title,values=shapes, labels=legend_label) +
  geom_hline(yintercept=0, linetype="dashed") +
  xlab("Program year") +
  ylab(yaxis_lab) +
  theme_complete_bw() +
  theme(strip.text.x = element_text(size = 14),
        axis.text.x = element_text(size = 12)) +
  ggtitle(title)
```

Imagine (or perhaps mournfully recall) the mess that can occur when you don't strictly style a complicated `ggplot` call. Trying to fix bugs and ensure your code is working can be a nightmare. Now imagine trying to do it with the same code 6 months after you've written it. Invest the time now and reap the rewards as the code practically explains itself, line by line.

5.2 Automated Tools for Style and Project Workflow

5.2.1 Styling

1. **Code Autoformatting** - RStudio includes a fantastic built-in utility (keyboard shortcut: `CMD-Shift-A`) for autoformatting highlighted chunks of code to fit many of the best practices listed here. It generally makes code more readable and fixes a lot of the small things you may not feel like fixing yourself. Try it out as a "first pass" on some code of yours that *doesn't* follow many of these best practices!

2. **Assignment Aligner** - A cool R package allows you to very powerfully format large chunks of assignment code to be much cleaner and much more readable. Follow the linked instructions and create a keyboard shortcut of your choosing (recommendation: `CMD-Shift-Z`). Here is an example of how assignment aligning can dramatically improve code readability:

```
# Before
OUSD_not_found_aliases <- list(
  "Brookfield Village Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Bro
  "Carl Munck Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Munck"),
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "East Oakland PRIDE Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Eas
  "EnCompass Academy" = str_subset(string = OUSD_school_shapes$schnam, pattern = "EnCompass"),
  "Global Family School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Global"),
  "International Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "In
  "Madison Park Lower Campus" = "Madison Park Academy TK-5",
  "Manzanita Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Manzan
  "Martin Luther King Jr Elementary" = str_subset(string = OUSD_school_shapes$schnam, pattern = "
  "PLACE @ Prescott" = "Preparatory Literary Academy of Cultural Excellence",
  "RISE Community School" = str_subset(string = OUSD_school_shapes$schnam, pattern = "Rise Commun
)

# After
OUSD_not_found_aliases <- list(
  "Brookfield Village Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Carl Munck Elementary"              = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Community United Elementary School" = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "East Oakland PRIDE Elementary"      = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "EnCompass Academy"                  = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Global Family School"               = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "International Community School"     = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Madison Park Lower Campus"          = "Madison Park Academy TK-5",
  "Manzanita Community School"         = str_subset(string = OUSD_school_shapes$schnam, pattern =
  "Martin Luther King Jr Elementary"   = str_subset(string = OUSD_school_shapes$schnam, pattern =
```

```
  "PLACE @ Prescott"                    = "Preparatory Literary Academy of Cultural Exc
  "RISE Community School"               = str_subset(string = OUSD_school_shapes$schnam
)
```

3. **StyleR** - Another cool R package from the Tidyverse that can be powerful and used as a first pass on entire projects that need refactoring. The most useful function of the package is the `style_dir` function, which will style all files within a given directory. See the function's documentation and the vignette linked above for more details.
    - **Note**: The default Tidyverse styler is subtly different from some of the things we've advocated for in this document. Most notably we differ with regards to the number of spaces before/after "tokens" (i.e. Assignment Aligner add spaces before `=` signs to align them properly). For this reason, we'd recommend the following: `style_dir(path = ..., scope = "line_breaks", strict = FALSE)`. You can also customize StyleR even more if you're really hardcore.
    - **Note**: As is mentioned in the package vignette linked above, StyleR modifies things *in-place*, meaning it overwrites your existing code and replaces it with the updated, properly styled code. This makes it a good fit on projects *with version control*, but if you don't have backups or a good way to revert back to the intial code, I wouldn't recommend going this route.

Chapter 6

Data Management

Contributors: Kunal Mishra, Jade Benjamin-Chung, Ben Arnold

2019-10-10: THIS CHAPTER IS A WORK IN PROGRESS (INCOMPLETE)

6.1 Data input/output (I/O)

6.1.1 `.RDS` vs `.RData` Files

One of the most common ways to load and save data in Base R is with the `load()` and `save()` functions to serialize multiple objects in a single `.RData` file. The biggest problems with this practice include an inability to control the names of things getting loaded in, the inherent confusion this creates in understanding older code, and the inability to load individual elements of a saved file. For this, we recommend using the RDS format to save R objects using `saveRDS()` and its complement `readRDS()`.

- **Note**: if you have many related R objects you would have otherwise saved all together using the `save` function, the functional equivalent with `RDS` would be to create a (named) list containing each of these objects, and saving it.
- **Note**: there is an important caveat for `.rds` files: they are not automatically backward compatible across different versions of R! So, while they are very useful in general, beware. See, for example, this thread on StackExchange. `.csv` files embed slightly less information (typically), but are more stable across different versions of R.

6.1.2 `.CSV` Files

Once again, the `readr` package as part of the Tidvyerse is great, with a much faster `read_csv()` than Base R's `read.csv()`. For massive CSVs (> 5 GB), you'll find `data.table::fread()` to be the fastest CSV reader in any data science language out there. For writing CSVs, `readr::write_csv()` and `data.table::fwrite()` outclass Base R's `write.csv()` by a significant margin as well.

6.1.3 Publishing public data

NEVER push a dataset into the public domain (e.g., GitHub, OSF) without first checking with Ben to ensure that it is appropriately de-identified and we have approval from the sponsor and/or human subjects review board to do so.

If you are releasing data into the public domain, then consider making available *at minimum* a `.csv` file and a codebook of the same name (note: you should have a codebook for internal data as well). We often also make available `.rds` files as well. For example, your `mystudy/data/public` directory could include three files for a single dataset, two with the actual data in `.rds` and `.csv` formats, and a third that describes their contents:

```
analysis_data_public.csv
analysis_data_public.rds
analysis_data_public_codebook.txt
```

In general, datasets are usually too big to save on GitHub, but occasionally they are small. Here is an example of where we actually pushed the data directly to GitHub: https://github.com/ben-arnold/enterics-seroepi/tree/master/data .

If the data are bigger, then maintaining them under version control in your git repository can be unweildy. Instead, we recommend using another stable repository that has version control, such as the Open Science Framework (osf.io). For example, all of the data from the WASH Benefits trials (led by investigators at Berkeley, icddr,b, IPA-Kenya and others) are all stored through data components nested within in OSF projects: https://osf.io/tprw2/.

6.2 Documenting datasets

Datasets need to have metadata (documentation) associated with them to help people understand them. Well documented datasets save an enormous amount of time because it helps avoid lots of back-and-forth with new people orienting themselves with the data. This applies to both private and public data used in your work flow.

Each final dataset should include a codebook. The file asembo_analysis_codebook.txt provides one example of what a codebook for a simple dataset could contain.

For complex studies with multiple, relational data files, it is exceptionally helpful to also include a README overview in plain text or markdown that explains the relationships between the datasets. Here is an example from the WASH Benefits Bangladesh trial primary outcomes analysis: README-WBB-primary-outcomes-datasets.md.

Chapter 7

GitHub and Version Control

Contributors: Stephanie Djajadi, Nolan Pokpongkiat, and Ben Arnold

7.1 Basics

Git is a version control system. It has very good documentation online: https://git-scm.com/doc

If you get into trouble with Git, the docs will help a lot!

Git is often used in conjunction with GitHub, which is an online platform to help teams collaborate while using Git for version control: https://github.com.

- A detailed tutorial of Git can be found here on UC Berkeley's CS61B website.
- If you are already familiar with Git, you can reference the summary at the end of Section B.
- If you have made a mistake in Git, you can refer to this article to undo, fix, or remove commits in git.

7.2 Git Branching

A terrific overview of branching workflow and its rationale is here: https://guides.github.com/introduction/flow/

Branches allow you to keep track of multiple versions of your work simultaneously, and you can easily switch between versions and merge branches together

once you've finished working on a section and want it to join the rest of your code. Here are some cases when it may be a good idea to branch:

- You may want to make a dramatic change to your existing code (called refactoring) but it will break other parts of your project. But you want to be able to simultaneously work on other parts or you are collaborating with others, and you don't want to break the code for them.
- You want to start working on a new part of the project, but you aren't sure yet if your changes will work and make it to the final product.
- You are working with others and don't want to mix up your current work with theirs, even if you want to bring your work together later in the future.

A detailed tutorial on Git Branching can be found here. You can also find instructions on how to handle merge conflicts when joining branches together.

7.3 Example Workflow

A standard workflow when starting on a new project and contributing code looks like this:

Command	Description
SETUP: FIRST TIME ONLY: `git clone <url> <directory_name>`	Clone the repo. This copies of all the project files in its current state on Github to your local computer.
1. `git pull origin master`	update the state of your files to match the most current version on GitHub
2. `git checkout -b <new_branch_name>`	create new branch that you'll be working on and go to it
3. Make some file changes	work on your feature/implementation
4. `git add <filename>`	add file to stage for commit
5. `git commit -m <commit message>`	commit file with a message
6. `git push -u origin <branch_name>`	push branch to remote and set to track (-u only works if this is first push)
7. Repeat step 4-5.	work and commit often
8. `git push`	push work to remote branch for others to view
9. Follow the link given from the `git push` command to submit a pull request (PR) on GitHub online	PR merges in work from your branch into master

Command	Description
(10.) Your changes and PR get approved, your reviewer deletes your remote branch upon merging	
11. `git fetch --all --prune`	clean up your local git by untracking deleted remote branches

Other helpful commands are listed below.

7.4 Commonly Used Git Commands

Command	Description
`git clone <url> <directory_name>`	clone a repository, only needs to be done the first time
`git pull origin master`	pull before making any changes
`git branch`	check what branch you are on
`git branch -a`	check what branch you are on + all remote branches
`git checkout -b <new_branch_name>`	create new branch and go to it (only necessary when you create a new branch)
`git checkout <branch name>`	switch to branch
`git add <file name>`	add file to stage for commit
`git commit -m <commit message>`	commit file with a message
`git push -u origin <branch_name>`	push branch to remote and set to track (-u only works if this is first push)
`git branch --set-upstream-to origin <branch_name>`	set upstream to origin/ (use if you forgot -u on first push)
`git push origin <branch_name>`	push work to branch
`git checkout --track origin/<branch_name>`	pulls a remote branch and creates a local branch to track it (use when trying to pull someone else's branch onto your local computer)
`git push --delete <remote_name> <branch_name>`	delete remote branch

Command	Description
`git branch -d <branch_name>`	deletes local branch, -D to force
`git fetch --all --prune`	untrack deleted remote branches

7.5 How often should I commit?

Stephanie and Nolan (trained in CS and Data Science) suggest it is good practice to commit every 15 minutes (a time-based guideline), or every time you make a significant change (progress-based guideline). Ben's perspective aligns with this view, but is weighted toward committing around completion of discrete chunks of work; for him, a discrete chunk of work will often take quite a bit longer than 15 minutes time. Take home message: *It is better to commit more rather than less.*

7.6 What should be pushed to Github?

In general, it is better to track text-based files (`.R`, `.Rmd`, `.md`,`.txt`,etc…) compared with binary files (`.pdf`,`.png`,`.docx`, etc…) because Git will store changes to a binary file as a completely new file in your Git directory. If you store 100 versions of the same binary file, your directory will quickly become very bloated. If the binary files don't change often, then you could consider including them under version control, but it is usually cleaner to keep them under a separate version control, such as through an Open Science Framework project with a specific data component.

Be careful before you push .Rout log files! If someone else runs an R script and creates an .Rout file at the same time and both of you try to push to github, it is incredibly difficult to reconcile these two logs. If you run logs, keep them on your own system or (preferably) set up a shared directory where all logs are name and date timestamped.

There is a standardized `.gitignore` for `R` which you can download and add to your project. This ensures you're not committing log files or things that would otherwise best be left ignored to GitHub. This is a great discussion of project-oriented workflows, extolling the virtues of a self-contained, portable projects, for your reference.

7.7 How should I describe my commit?

When you commit, always include a short commit message that describes what the commit does. In the command line, you can achieve this after you have staged files to commit with the `commit -m <"your commit message here">` syntax. This helps track-back through work flow. For example, if the commit message is `new`, that doesn't provide any information about what the commit includes. A more descriptive commit message would be `Create first draft of Fig 1 distribution plot` or `Change color scheme for distribution plot`.

For more lengthy and detailed commit messages, which go beyond the simple, single line `commit -m <your commit message here>` synatax, this Medium post on the anatomy of a good commit message includes additional discussion. (Note, we don't typically use commits that are this detailed!)

Chapter 8

Working with Big Data

Contributors: Eric Kim, Kunal Mishra and Jade Benjamin-Chung

8.1 Basics

A pitfall of working in R is that all objects are stored in memory - this makes it very difficult to work with datasets that are larger than 1-2 Gb for most standard computers. Here, we'll explore some alternatives to working with big data.

The Berkeley Statistical Computing Facility also has many good training resources.

8.2 Using downsampled data

In studies with very large datasets, we save "downsampled" data that usually includes a 1% random sample stratified by any important variables, such as year or household id. This allows us to efficiently write and test our code without having to load in large, slow datasets that can cause RStudio to freeze. Be very careful to be sure which dataset you are working with and to label results output accordingly.

8.3 Unix

Though bash is very commonly used for management of your file system (see Chapter 9), it is also a very capable at doing basic data manipulation with big

data. At the core, since the data is stored on disk, you avoid having to overload memory when using bash commands as it will work with the files directly. By default, these commands will print the results to standard output (probably your terminal screen) and you can then redirect the results to other files on disk. These commands can also be chained via pipes (represented as `|`, similar to `%>%` in tidyverse). All of these have a list of arguments that can be passed in via flags (check the `man` page for more details on each).

Command	Description
`head`/`tail`	Displays the first few or last few rows of a file
`cat`	Concatenates files and prints them
`sort`	Sorts the file
`cut`	Cuts out portions of each line and prints it
`grep`	Finds lines of a file that matches inputted patterns
`sed`	Find and replace
`awk`	Similar to `grep` and `sed` but with some extra programmatic functionality
`uniq`	Unifies repeated lines (combine with `sort` to get unique rows)
`wget` / `curl`	Downloads data/files from websites

8.4 SQL and `dbplyr`

SQL databases are relational databases that are a collection of *tables* that consists of *fields* or *attributes*, each containing a single *type*. If you use `dplyr` a lot, you will find that it is heavily inspired with a SQL flavor in mind. Formally, data gets loaded onto a database system and it is stored on disk. This alone makes working with data fast, but the real efficiency gain is the concept of indexing. If you are curious, most SQL databases implement their index with B trees or B+ trees, which allow for log time complexity for search operations in average and worst case scenarios while providing constant time complexity in best case scenario.

The basic structure of a SQL query is as follows:

```
SELECT [DISTINCT] (attributes)
FROM (table)
[WHERE (conditions)]
[GROUP BY (attributes) [HAVING (conditions)]]
[ORDER BY (attributes) [DESC]]
```

The equivalent `dplyr` command would look as such:

```
table %>%
  select(attributes) %>%     # distinct(attributes) for select distinct
  group_by(attributes) %>%   #
  filter(conditions) %>%     #
  arrange(attributes)        # arrange(desc(attributes)) for descending
```

There is ample support for connection to databases in R, and, in particular, there is the `dbplyr` package, which allows you to interface with the data with `dplyr` code instead of SQL code.

8.5 data.table and dtplyr

It is often possible to load large datasets into memory in R, however computations will probably be very slow. One way around this is to use `data.table`. You will find that operations on data are much faster than base R or `dplyr` even though data is loaded into memory - this is because of clever programming in C as well as internally creating a *key* (the SQL equivalent of an index) by default when loading in the data. You can improve on this even more by setting extra keys for variables you know you will be doing filter or join operations on.

More recently from the tidyverse, is the implementation of `dtplyr`, which allows for `dplyr` syntax on `data.table` objects.

An overview of the `dplyr` vs `data.table` debate can be found in this stackoverflow post and all 3 answers are worth a read.

8.6 ff, bigmemory, biglm

Sometimes, it may be impossible to load data into memory. Because of the overhead required, you can expect around twice as much memory needed as the size of the file on disk to just load in a sufficiently large dataset. One way to work around this is to keep the data on disk and instead create clever data structures that allow for natural interfacing with the data in an R session while mapping operations to the data on disk. Two packages that implement these ideas are `ff` and `bigmemory`.

We can now interface with the data while avoiding loading it into memory but we run into issues when we try to fit models on it. For an $n \times p$ dataset, linear regression has a time complexity of $O(np^2 + p^3)$ and a space complexity of $O(np + p^2)$ (this just means it will take a while and take up a lot of space for large n and even moreso for large p). Clever solutions used in machine learning (think iterative algorithms stochastic gradient descent) can help us here. The idea is totake a smaller portion of our data, fit regression then update the

coefficient based on another run of linear regression on another small portion of the data. For GLM models, this can be done with the `biglm` package which has integration with `ff` and `bigmemory`.

8.7 Parallel computing

8.7.1 Embarassingly Parallel Problems

Sometimes, we have to do something in a loop-like structure where each iteration may be independent of each other (think, simulations or bootstrap). These types of loops are referred to as *embarassingly parallel* problems (I did not make this up). Each iteration may take some time and every iteration thereafter must wait because the loop is essentially operation as a queue. This is where parallel computing comes into play: every computer these days come with at least 2 cores in the CPU. Each CPU core can operate independently so after some overhead, we can speed up our loop by the number of cores our computers have.

8.7.2 Packages

In R, the popular packages are `parallel`, `foreach`, and `doParallel` (the back-end that connects `foreach` and `parallel`). More modern parallel computing packages in R are `future` and `furrr` (inspired by `future purrr`, it allows for `purrr` like syntax using the `future` data structure from its namesake package). In Python, the `Dask` package has similar functionality to `future`. Note: the `parallel` package comes with a `detectCores` function, but I sometimes find that it is not accurate. On a mac, you can manually check the number of cores by going into About This Mac then System Report then checking the Total Number of Cores in the Hardware tab.

8.7.3 GPU's

For most everyday tasks, CPU will be sufficient, but for large problems even an 8x speed boost (for a computer with 8 cores) might not be enough. This is where GPU's come into play. While CPU cores are good at complex operations, GPU cores are good at many small operations (think matrix multiplication). As GPU cores come in the hundreds for cheaper graphics cards and thousands for top end graphics cards, they are ideal for training machine learning models, particularly neural networks. However, as these GPU cores were intended for the graphics on our computers, we cannot access their computing power out of R or Python without some translation in between. Graphics manufacturers have been catching up to this market, and one of the most popular platforms

for parallel computing on GPU's is Nvidia's CUDA for use with Nvidia graphics cards.

8.7.4 The *MapReduce* paradigm

The idea of the *MapReduce* paradigm is that we can distribute the data across many nodes and try to do the computation on each piece of the data in each node. One benefit of this is that if our data is too large to fit on disk for a single machine, we can instead spread it across many then do our operation on parallel and aggregate it back together. We can formalize this paradigm into three steps

- Map: Split the data into sub-datasets and perform an operation on each entry in each sub-dataset thereby creating key-value pairs.
- Shuffle: Merge the key-value pairs and sort them.
- Reduce: Apply an operation on the associated values for each key.

An excellent example is included at the bottom of this link. A similar paradigm that is implemented in the tidyverse is the *split-apply-combine* strategy.

The popular infrastructures for doing parallel computing with the MapReduce paradigm are Hadoop and Spark (think of Spark as an in-memory version of Hadoop). Spark can be interfaced with through Python via PySpark or R via SparkR (from Apache) or sparklyr (from RStudio), however note that Spark is natively implemented in Java and Scala so the overhead of serialization between R/Python to Java/Scala may be a time expensive operation.

8.8 Optimal RStudio set up

Using the following settings will help ensure a smooth experience when working with big data. In RStudio, go to the "Tools" menu, then select "Global Options". Under "General":

Workspace

- **Uncheck** Restore RData into workspace at startup
- Save workspace to RData on exit – choose **never**

History

- **Uncheck** Always save history

Unfortunately RStudio often gets slow and/or freezes after hours working with big datasets. Sometimes it is much more efficient to just use Terminal / gitbash to run code and make updates in git.

Chapter 9

UNIX Commands

Contributors: Stephanie Djajadi, Kunal Mishra, Anna Nguyen, Jade Benjamin-Chung, and Ben Arnold

We typically use Unix commands in Terminal (for Mac users) or Git Bash (for Windows users) to

1. Run a series of scripts in parallel or in a specific order to reproduce our work
2. To check on the progress of a batch of jobs
3. To use git and push to github

9.1 Environment

On Mac OS, there is an application named `Terminal` that provides a bash shell interface to Unix. In Windows, one option is to install the `git for Windows` package: https://gitforwindows.org/.

The default coloring in a terminal window is pretty basic. If you want to make it more colorful in Mac OS, you can do that by saving a `.bash_profile` file in your home directory (note the "." prefix on the file name). This is one example of how you can add color to your terminal by including custom coloring in your bash profile (copied from Ben's profile):

```
# color terminal
export CLICOLOR=1
export LSCOLORS=GxFxCxDxBxegedabagaced
export PS1='\[\033[01;32m\]\u@\h\[\033[00m\]:\[\033[01;34m\]\w\[\033[00m\]\$ '
```

The encoding is extremely cryptic, but there are decodings online (e.g., link).

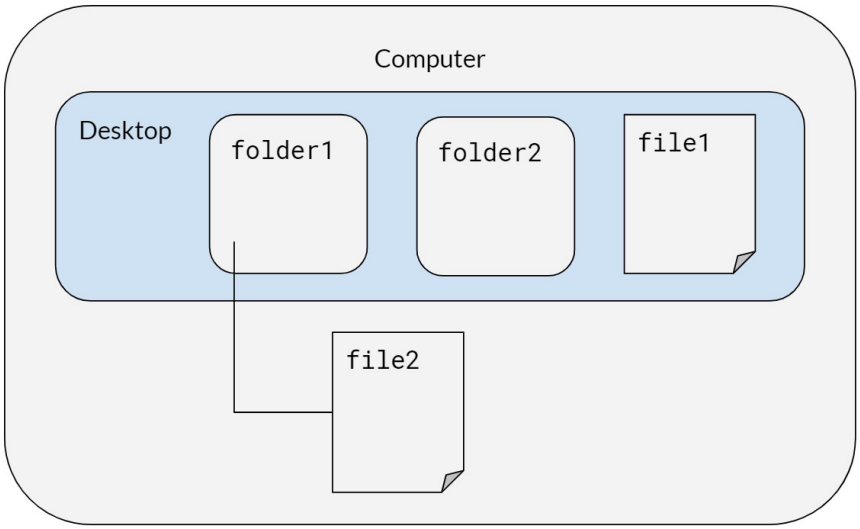


Figure 9.1: Here is our example desktop.

Another bash shell that provides a large array of colors for Mac OS is iTerm2 (https://iterm2.com/). There are over 200 color schemes to choose from: https://github.com/mbadolato/iTerm2-Color-Schemes.

9.2 Basics

On the computer, there is a desktop with two folders, `folder1` and `folder2`, and a file called `file1`. Inside `folder1`, we have a file called `file2`. Mac users can run these commands on their terminal; we recommend that Windows users use Git Bash, not Windows PowerShell.

9.3 Syntax for both Mac/Windows

When typing in directories or file names, quotes are necessary if the name includes spaces.

Command	Description
`cd desktop/folder1`	Change directory to `folder1`
`pwd`	Print working directory
`ls`	List files in the directory

Command	Description
`cp "file2" "newfile2"`	Copy file (remember to include file extensions when typing in file names like `.pdf` or `.R`)
`mv "newfile2" "file3"`	Rename `newfile2` to `file3`
`cd ..`	Go to parent of the working directory (in this case, `desktop`)
`mv "file1" folder2`	Move `file1` to `folder2`
`mkdir folder3`	Make a new folder in `folder2`
`rm <filename>`	Remove files
`rm -rf folder3`	Remove directories (`-r` will attempt to remove the directory recursively, `-rf` will force removal of the directory)
`clear`	Clear terminal screen of all previous commands

9.4 Running Bash Scripts

Windows	Mac / Linux	Description
`chmod +750 <filename.sh>`	`chmod +x <filename.sh>`	Change access permissions for a file (only needs to be done once)
`./<filename.sh>`	`./<filename.sh>`	Run file (`./` to run any executable file)
`bash bash_script_name.sh &`	`bash bash_script_name.sh &`	Run shell script in the background

9.5 Running Rscripts in Windows

Note: This code seems to work only with Windows Command Prompt, not with Git Bash.

When R is installed, it comes with a utility called Rscript. This allows you to run R commands from the command line. If Rscript is in your `PATH`, then typing Rscript into the command line, and pressing enter, will not error. Otherwise, to use Rscript, you will either need to add it to your PATH (as an environment variable), or append the full directory of the location of Rscript on your machine. To find the full directory, search for where R is installed your computer. For

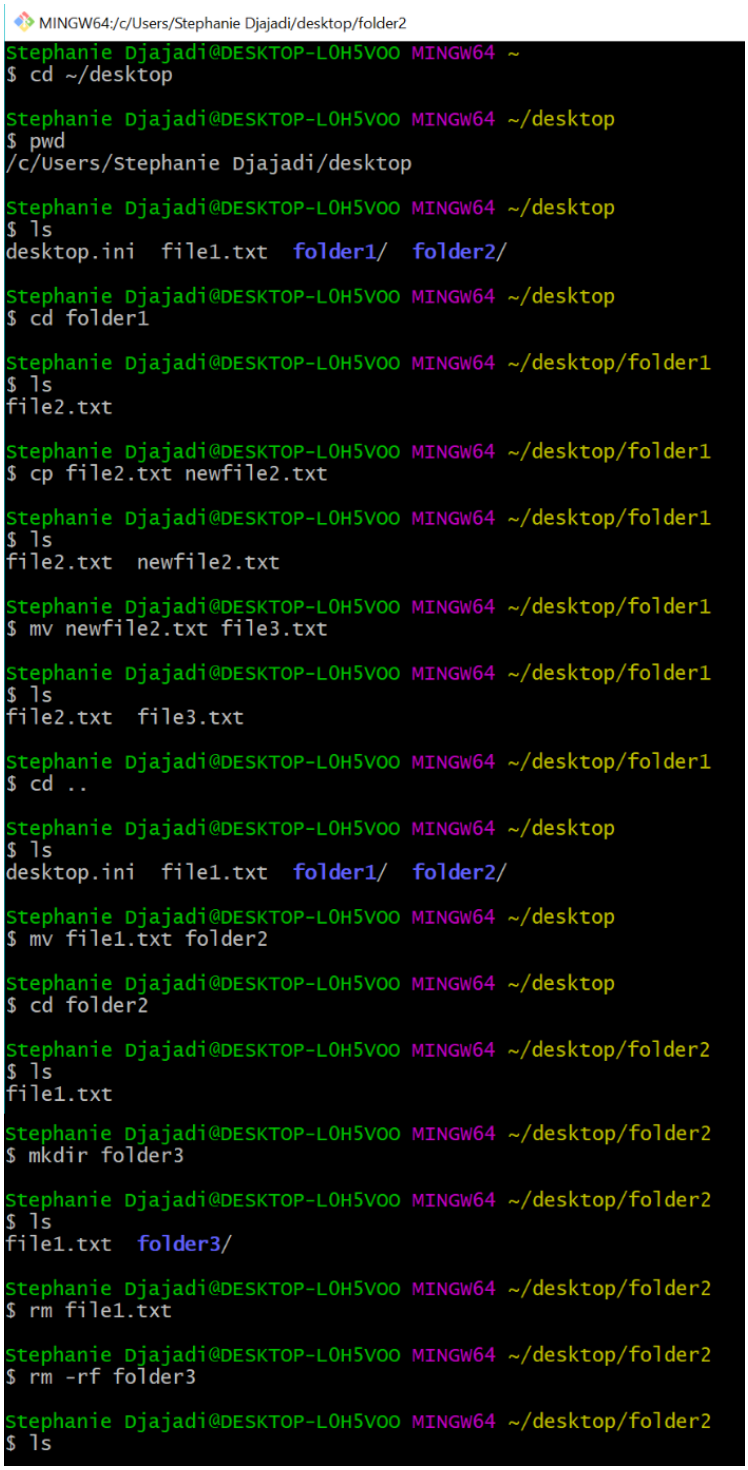


Figure 9.2: Here is an example of what your terminal might look like after executing the commands in the order listed above.

instance, it may be something like below (this will vary depending on what version of R you have installed):

`C:\Program Files\R\R-3.6.0\bin`

For appending the `PATH` variable, please view this link. I strongly recommend completing this option.

If you add the PATH as an environment variable, then you can run this line of code to test: `Rscript -e "cat('this is a test')"`, where the `-e` flag refers to the expression that will be executed.

If you do not add the PATH as an environment variable, then you can run this line of code to replicate the results from above: `"C:\Program Files\R\R-3.6.0\bin" -e "cat('this is a test')"`

To run an R script from the command line, we can say: `Rscript -e "source('C:/path/to/script/some_code.R')"`

9.5.1 Common Mistakes

- Remember to include all of the quotation marks around file paths that have a spaces.
- If you attempt to run an R script but run into `Error: '\U' used without hex digits in character string starting "'C:\U"`, try replacing all \ with \\ or /.

9.6 Checking tasks and killing jobs

Windows	Mac / Linux	Description
`tasklist`	`ps -v`	List all processes on the command line
	`top -o [cpu/rsize]`	List all running processes, sorted by CPU or memory usage
`taskkill /F /PID pid_number`	`kill <PID_number>`	Kill a process by its process ID
`taskkill /IM "process name" /F`		Kill a process by its name
`start /b program.exe`		Runs jobs in the background (exclude `/b` if you want the program to run in a new console)

Windows	Mac / Linux	Description
	`nohup`	Prevents jobs from stopping
	`disown`	Keeps jobs running in the background even if you close R
`taskkill /?`		Help, lists out other commands

To kill a task in Windows, you can also go to Task Manager > More details > Select your desired app > Click on End Task.

9.7 Running big jobs

For big data workflows, the concept of "backgrounding" a bash script allows you to start a "job" (i.e. run the script) and leave it overnight to run. At the top level, a bash script (`0-run-project.sh`) that simply calls the directory-level bash scripts (i.e. `0-prep-data.sh`, `0-run-analysis.sh`, `0-run-figures.sh`, etc.) is a powerful tool to rerun every script in your project. See the included example bash scripts for more details.

- **Running Bash Scripts in Background**: Running a long bash script is not trivial. Normally you would run a bash script by opening a terminal and typing something like `./run-project.sh`. But what if you leave your computer, log out of your server, or close the terminal? Normally, the bash script will exit and fail to complete. To run it in background, type `./run-project.sh &; disown`. You can see the job running (and CPU utilization) with the command `top` or `ps -v` and check your memory with `free -h`.

Alternatively, to keep code running in the background even when an SSH connection is broken, you can use `tmux`. In terminal or gitbash follow the steps below. This site has useful tips on using `tmux`.

```
# create a new tmux session called session_name
tmux new -ssession_name

# run your job of interest
R CMD BATCH myjob.R &

# check that it is running
ps -v

# to exit the tmux session (Mac)
```

```
ctrl + b
d

# to reopen the tmux session to kill the job or
# start another job
tmux attach -tsession_name
```

- **Deleting Previously Computed Results**: One helpful lesson we've learned is that your bash scripts should remove previous results (computed and saved by scripts run at a previous time) so that you never mix results from one run with a previous run. This can happen when an R script errors out before saving its result, and can be difficult to catch because your previously saved result exists (leading you to believe everything ran correctly).
- **Ensuring Things Ran Correctly**: You should check the `.Rout` files generated by the R scripts run by your bash scripts for errors once things are run.

Chapter 10

Communication and Coordination

Contributors: Jade Benjamin-Chung, Ben Arnold

These communications guidelines are evolving as we increasingly adopt Slack, but here some general principles if you work closely with Ben.

10.1 Slack

- If you work with Ben but are not a member of Ben's Slack workspace then ask him to invite you!
- Use Slack for scheduling, coding related questions, quick check ins, etc. If your Slack message exceeds 200 words, it might be time to use email.
- Use channels instead of direct messages unless you need to discuss something private.
- Include tags on your message (e.g., `@Ben`) when you want to ensure that a person sees the message. Ben doesn't regularly read messages where he isn't tagged.
- Please make an effort to respond to messages that message you (e.g., `@Ben`) as quickly as possible and always within 24 hours, unless of course you are on vacation!
- If you are unusually busy (e.g., taking MCAT/GRE, taking many exams) or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual and also set your status in Slack

(e.g., it could say "On vacation") so we know not to expect to see you online.

- Please thread messages in Slack as much as possible.

10.2 Email

- Use email for longer messages (>200 words) or messages that merit preservation.
- Generally, strive to respond within 24 hours. If you are unusually busy or on vacation please alert the team in advance so we can expect you not to respond at all / as quickly as usual.

10.3 Trello

- Ben manages projects and teams using a kanban board approach in Trello.
- You and/or Ben will add new cards within our team's Trello boards and assign them to team members.
- Each card represents a discrete chunk of work.
- Cards higher in a list are higher priority.
- Strive to complete the tasks in your card by the card's due date. Talk to Ben about deadlines – we can always manage the calendar!
- Use checklists to break down a task into smaller chunks. Usually, you can do this yourself (but ask Ben if you ever want input).
- Move cards to the "DONE" list on a board when they are done.

10.4 Google Drive

- We mostly use Google Drive to create shared documents with longer descriptions of tasks. These documents are linked to Trello cards. Ben often shares these docs with a whole project team since tasks are overlapping, and even if a task is assigned to one person, others may have valuable insights.
- Please invite both of Ben's email addresses to any documents you create (bfarnold@gmail.com, ben.arnold@ucsf.edu).

10.5 Calendar / Meetings

- Ben will schedule most meetings through the calendar.
- Our meetings start on the hour.

- If you are going to be late, please send a message in our Slack channel.
- If you are regularly not able to come on the hour, notify the team and we might choose the modify the agenda order or the start time.
- Add hoc meetings are welcome. If Ben's office door is open, come in!

Chapter 11

Code of conduct

Contributors: Jade Benjamin-Chung, Ben Arnold

11.1 Group culture

We strive to work in an environment that is collaborative, supportive, open, and free from discrimination and harassment, per University policies.

We encourage students / staff of all experience levels to respectfully share their honest opinions and ideas on any topic. Our group has thrived upon such respectful honest input from team members over the years, and this document is a product of years of student and staff input (and even debate) that has gradually improved our productivity and overall quality of our work.

If Ben is your PI, be forewarned that he tends to batch his email communication (~30 mins in the morning and afternoon, 15 mins mid-day), and doesn't tend to answer Slack or email during evenings or weekends. If you need to reach him urgently then give him a call or text on his mobile.

11.2 Protecting human subjects

All lab members must complete CITI Human Subjects Training and share their certificate with Ben. We will will add team members to relevant Institutional Review Board protocols to ensure they have permission to work with identifiable datasets.

One of the most relevant aspects of protecting human subjects in our work in the Data Coordinating Center is maintaining confidentiality and data privacy. For students supporting our data science efforts, in practice this means:

- If you are using a virtual computer (e.g., Google Cloud, AWS, Optum), never save the data in that system to your personal computer or any other computer without prior permission.
- Do not share data with anyone without first obtaining permission, including to other members of the Proctor Foundation, who might not be on the same IRB protocol as you (check with Ben or the relevant PI first).
- **NEVER** push a dataset into the public domain (e.g., GitHub, OSF) without first checking with Ben to ensure that it is appropriately de-identified and we have approval from the sponsor and/or human subjects review board to do so.

Remember, data that looks like it does not contain identifiers to you might still be classified as data that requires special protection by our IRB or under HIPAA, so always proceed with caution and ask for help if you have any concerns about how to maintain study participant confidentiality. For example, the combination of age, sex, and geographic location of the individual's town or neighborhood is typically considered identifiable.

11.3 Authorship

We adhere to the ICMJE Definition of authorship and are happy for team members who meet the definition of authorship to be included as co-authors on scientific manuscripts.

11.4 Work hours

Please follow the Proctor Foundation's employee guidelines for work hours, and discuss the specifics with your PI. If Ben is your PI, then work with him on your schedule to ensure we have overlap in the office and that you are around at key times for group meetings, etc.

Chapter 12

Additional Resources

TBD