Case Studies in Reproducible Research: a spring seminar at UCSC

Eric C. Anderson, Kristen C. Ruegg, Tina Cheng, and the students of EEB 295

2017-05-01

Contents

Chapter 1

Course Overview

This is the home of the notes for a proposed course in data analysis and reproducible research using R, Rstudio, and GitHub.

The seminar is called, "Case Studies in Reproducible Research," but we utter that title with the caveat that, although the organizers have quite a few case studies they could spin up for this course, the case studies we will be studying in this course are going to be actual research projects that *you*—the participants—are working on. You're gonna bring 'em, and we are going to collectively help you wrassle them into a reasonable and reproducible data analysis. In the process we will touch on a number of elements of data analysis with R.

We will be working through a healthy chunk of the material in Garrett Grolemund and Hadley Wickham's book, R for Data Science, which is readable for free at the link above. We intend to use a handful of our own data sets each week to illustrate points from the book and show the material in action on real data sets.

This is not intended as a "first course in R". Students coming to the course should have at least a modicum of familiarity with using R, and we will launch more directly into using the tools of the tidyverse. EEB students with little or no experience in R might be interested in sitting in with Giacomo Bernardi's lab group on Mondays at 3PM in the COH library. They are conducting a Bio 295 seminar, working through "a super basic book that takes the very first steps into R."

For the interested, these materials were all prepared using RStudio's bookdown package. The RStudio project in which it lives is hosted on eriq's GitHub page here

1.1 Meeting Times, Location, Requirements

Intended to be Friday afternoons, 1:45–3:15 PM in the library/conference room at Long Marine Lab.

Students must bring a laptop to do examples during the seminar, and all students are expected to have a data set that they are in the midst of analyzing (or upon which they hope to commence analysis soon!) for a research project. We will

1.2 The origin of this seminar

The idea for this course was floated by Tina Cheng who was planning to lead a seminar in spring 2017 based in part on Eric C. Anderson's "Reproducible Research Course", taught at the Southwest Fisheries Science Center in the fall of 2014. Although going over those notes might have been a reasonable exercise, it turns out that a lot has changed in the world of data analysis since fall 2014, and the notes from that course are, today, a little bit dated.

We have been particularly excited by the ascendancy of Hadley Wickham's tidyverse approach to data analysis, and the tremendous development of a variety of tools developed by RStudio for integrating report generation and data analysis into reproducible workflows. In fact, Eric has been saying for the last year that if he were to teach another course on data analysis it would be structured completely differently than his "Reproducible Research Course". So, it was clearly time for him to stop talking and help put together an updated and different course.

At the same time, in working on our own projects and in helping others, we have consistently found that the most effective way for anyone to learn data analysis is to ensure that it is immediately relevant to whatever ongoing research project is currently consuming them. Therefore, in the current seminar, we are hoping to spend at least half of our time "workshopping" the data sets that seminar participants are actually involved in analyzing. Together we will help students wrestle their data, analyses, and ideas into a single, well-organized RStudio project under version control with git. Therefore, every student should come to this course with a data set and an associated analysis project.

1.3 Course Organizers

Kristen C. Ruegg Kristen is a conservation geneticist who specializes in the application of genome-wide data to understand population level processes and inform management, with a particular focus on migratory birds. She has has been enlightened to the powers of the "tidyverse" over the last couple of years (mostly through the constant insistence of her enthusiastic husband Eric Anderson) and is looking forward to becoming more fluid in its application over the course of the quarter. Her main role in this course will be to help with the course design and logistics and help reign Eric in when he has started to orbit into some obscure realm of statistical nuance.

Eric C. Anderson Eric trained as a statistician who specializes in genetic data. Since 2003 he has worked at the NMFS Southwest Fisheries Science Center in Santa Cruz. Although much of his statistical research involves the development of computationally intensive methods for specialized analyses of genetic data, he has been involved in a variety of data analysis projects at NMFS and with collaborators worldwide. Eric was an early adherent to reproducible research principles and continues, as such, performing most of his research and data analysis in the open and publicly available on GitHub (find his GitHub page here). In 2014, he taught the "Reproducible Research Course" at NMFS, and is excited to provide an updated version, focusing more, this time, on the recently developed "tidyverse".

Tina Cheng Tina is a graduate student in EEB. She is going to be leading the session during the first week of the course when Kristen and Eric are still on spring break, and then she is going to be joining in on the fun with us for the remainder of the quarter until she has to travel off to Baja, TA-ing the "supercourse" during the last four weeks of the quarter.

1.4 Course Goals

The goal of this course is for scientists, researchers, and students to learn to:

- properly store, manage, and distribute their data in a *tidy* format
- consolidate their digital research materials and analyses into well-organized RStudio projects.
- use the tools of the tidyverse to manipulate and analyze those data sets
- integrate data analysis with report generation and article preparation using the Rmarkdown format and using R Notebooks
- use git version control software and GitHub to effectively manage data and source code, collaborate efficiently with other researchers, and neatly package their research.

By the end of the course, the hope is that we will all have mastered strategies allowing us to use the above-listed, freely-available and open-source tools for conducting research in a reproducible fashion. The ideal we will be striving for is to be able to start from a raw data set and then write a computer program that

conducts all the cleaning, manipulation, and analysis of the data, and presentation of the results, in an automated fashion. Carrying out analysis and report-generation in this way carries a number of advantages to the researcher:

1. Newly-collected data can be integrated easily into your analysis.
2. If a mistake is found in one section of your analysis, it is not terribly onerous to correct it and then re-run all the downstream analyses.
3. Revising a manuscript to address referee comments can be done quickly.
4. Years after publication, the exact steps taken to analyze the data will still be available should anyone ask you how, exactly, you did an analysis!
5. If you have to conduct similar analyses and produce similar reports on a regular bias with new data each time, you might be able to do this readily by merely updating your data and then automatically producing the entire report.
6. If someone finds an error in your work, they can fix it and then easily show you exactly what they did to fix it.

Additionally, packaging one's research in a reproducible fashion is beneficial to the research community. Others that would like to confirm your results can do so easily. If someone has concerns about exactly how a particular analysis was carried out, they can find the precise details in the code that you wrote to do it. Someone wanting to apply your methods to their own data can easily do so, and, finally, if we are all transparent and open about the methods that we use, then everyone can learn more quickly from their colleagues.

In many fields today, publication of research requires the submission of the original data to a publicly-available data repository. Currently, several journals require that all analyses be packaged in a clear and transparent fashion for easy reproduction of the results, and I predict that trend will continue until most, if not all, journals will require that data analyses be available in easily reproduced formats. This course will help scientists prepare themselves for this eventuality. In the process, you will probably find that conducting your research in a reproducible fashion helps you work more efficiently (and perhaps even more enjoyably!)

1.5 Weekly Syllabus

1.5.1 Week 1 — Introduction and Getting Your Workspace Set Up

- At the end of this session we want to make sure that everyone has R, RStudio, and Git installed on their systems, and that they are working as expected.
- Additionally, everyoneshould have a free account on GitHub.
- And finally we need everyone's email address.

Some things to do:

- Get Rstudio cheat Sheets!
- Assemble data into a project
- Get private GitHub repos

Eric! You need to make an example project repo.

1.5.2 Week 2 — RStudio project organization; using git and GitHub; Quick RMarkdown

After this, students are going to have to put their own data into their own repositories and write a README.Rmd and make a README.md out of it.

1.5.3 Week 3 — Tibbles. Reading data in. Data rectangling

- Reading data into the data frames.
- read.table and read.csv
- tibbles
- The readr package
- Data types in the different columns and quick data sanity checks.
- A few different gotcha's
- Saving and reading data in R formats. `saveRDS` and `readRDS`.

1.5.4 Week 4 —

Chapter 2

Week One Meeting

Tina is going to be helping everyone get their systems all set up. After that we will have everyone clone an RStudio project from GitHub to see how easy that is.

2.1 Software Installation

1. **RStudio:** We want the latest "development" version of RStudio becuase it has features that we may want to use during this course. Get it from https://www.rstudio.com/products/rstudio/download/preview/ and install the appropriate one for your OS.
2. **R:** Let's make sure that we are all using the latest version of R. On March 7, 2017, version 3.3.3 was released. Go to https://cran.r-project.org/ and find the download link for your computer system. Download it and install it.
3. **bookdown:** This package is what I used to create these course notes. Getting it automatically installs a lot of other packages that are useful for authoring reproducible research. We want the latest development version, which can be obtained from GitHub by issuing the following commans at the R prompt (i.e. in the console window of RStudio:)

```
install.packages("devtools")
devtools::install_github("rstudio/bookdown")
```

4. Install **other packages** that we are going to be needing in the first few weeks. If you don't know how to install packages, ask Tina and she can show you. Install: `tidyverse`, and `stringr`.
5. Make sure that **git** is up and running on your system.
   - If you are using a Mac with a reasonably new OS, you should be able to just open the Terminal application (`/Applications/Utilities/Terminal`) and type "git" at the command line. If you have `git` it will say something that starts like:

```
    usage: git [--version] [--help] [-C <path>] [-c name=value]
   [--exec-path[=<path>]] [--html-path] [--man-path] [--info-path]
   [-p | --paginate | --no-pager] [--no-replace-objects] [--bare]
   [--git-dir=<path>] [--work-tree=<path>] [--namespace=<name>]
   <command> [<args>]

These are common Git commands used in various situations:

start a working area (see also: git help tutorial)
```

```
clone      Clone a repository into a new directory
etc. etc. etc.
```

If you do not have `git` then it should pop up a little thing asking if you would like to install a reduced set of developer tools. You do. Click OK. **NOTE** Instead of a pop up it might say something like, "xcrun Error: invalid active developer path. etc. etc…". In that case, you can install a fresh set of command line tools by typing this at the command line:

```
xcode-select --install
```

- If you are using a PC, I can't be as much help, but you can find links with instructions on how to download `git` for a PC here.
- If you are using Linux then we will assume you know how to get `git` or that you already have it.

2.2 Get an account on GitHub

If you don't already have an account on GitHub, go to github.com and click the "sign up" link near upper right of the page. It is pretty self-explanatory. Go ahead and get a **free** account. There is nothing to pay for here!

2.2.1 Private repositories

If you are a graduate student and you do not feel comfortable posting your data on a public site like GitHub, then you should request some private repositories from GitHub. GitHub has a great deal for academic users like students: free private repositories. Please go to https://education.github.com/pack to sign up for your free student pack.

2.3 Open an RStudio Project from GitHub

I am going to have everyone use RStudio and GitHub to clone and open an RStudio project that I prepared as a template so that people can see how I would like them to start putting together their own projects.

To open this project, from RStudio, go to the menu option "File->New Project…". Then from the resulting dialog, choose "Version Control". Then choose "Git". Then it asks for a "repository URL". Supply this: `https://github.com/eriqande/rep-res-coho-example` and leave the "Project Directory Name" empty. And then choose a directory in which to put it and click OK.

Bam! That will pull the RStudio project off of GitHub, make a local clone of it on your hard drive and open.

Once you have done that. Open `README.Rmd` within the project, and click the "knit" button which should be present near the top left of the editor window.

That is how you convert an R Markdown README to README.md which is easy to read and see on GitHub.

If you want to see what the project repository looks like on GitHub, have a look at https://github.com/eriqande/rep-res-coho-example.

2.4 Assignment for next week: Create an RStudio Project with Your Own Data

Your mission for the following week—i.e., please have this done (or as done as you can get it) by Friday, April 14, 2017—is to prepare an RStudio project with your own data set, and provide some background about the data and the ways that you would like to analyze it. The "rep-res-coho-example" is an example of what I have in mind for this. You should use the README.Rmd from that project as a template for your own README.Rmd. (To do this you can just copy the README.Rmd file into the top level of your project directory and then edit it to reflect your own data and project.)

To do all this you are going to want to make your own project. Do that like this:

1. In RStudio, choose "File->New Project…"
2. Then choose "New Directory" and then choose "Empty Project"
3. In the next dialog, choose a name (*it is best to use only letters, numbers, dashes, and underscores, and include no spaces in the name*) for it **and be sure to click the "Create a git repository" button**.
4. Then click "Create Project".

That should give you a new project. Here are some guidelines for putting your own data in there

- Put all of your data in a directory named `data` in your project.
- CSV (comma separated values) is probably the best format to use. It is text-readable without proprietary software (unlike an Excel file); however if you need to look at it in a tabular way with Excel, (gasp), you can do that easily. Tab-delimited text works if you have that, but CSV is preferred.
- Use only letters, numbers, dashes, and underscores for the file names, (and periods for their extensions, i.e., `.csv`)
- Give a brief description of your data in the README.Rmd.

2.5 Reading for next week

This week (before Friday, April 14, 2017), please read the following sections of the R for Data Science book

- Workflow basics: super basic review on how R works.
- Workflow: projects: info about organizing RStudio projects.
- Workflow: scripts: how to evaluate code in scripts.
- tibbles: a streamlined data frame format.
- data import This is our key reading for the week.

When you are done with the *Data Import* reading, take a whack at writing some code to read the data files in your project into a variable (or several variables).

Chapter 3

Week Two Meeting

For this week, everyone should have completed the reading listed in Section **??**. And everyone should have at least been trying to set up an RStudio Project with their own data in it, and given a whack at reading their data into a variable.

3.1 Workflow and Project Recap

3.1.1 Workflow: basics

3.1.1.1 Style

I just want to reiterate a few things that might seem minor, but are stylistically important in the long run.

1. Don't use `=` for assignment. Use `<-`. On a Mac use Option-"-" to get that more quickly.
   - Hey! Note that you *do* use `=` (and **only** `=`) for *passing values to function arguments*.

```
# do this:
my_variable <- 10

# don't do this (it works but is not good style)
my_variable = 10


# do this
result <- my_function(arg1 = 10, arg2 = "foo")

# don't do this (it might return a results, but will likely be incorrect)
result <- my_function(arg1 <- 10, arg2 <- "foo")

# for example, figure out how this fails:
matrix(data <- 1:10, nrow <- 5, byrow <- TRUE)
```

2. Put spaces around both sides of `=`, and `<-`, and other mathematical operators like `+`, `-`, `*`, etc.
3. Put spaces after commas.
4. Use R-Studio's magical `Cmd-i` keyboard shortcut to automatically indent highlighted code.
5. If you want to really geek out, Hadley shares his style tips more completely in his Advanced R Programming book.

This might seem pedantic, but adhering to these conventions makes it much easier for people to read your code.

3.1.1.2 TAB-completion

This is HUGE!! Start developing a twitchy left pinky now!

TAB early; TAB often!

Note that that completions of R code in the console or in the source window are context dependent:

- variables in global environment
- Functions
- Quoted strings complete to filenames in directories
- Installed packages in `library()`
- Help topics after `?`
- Function arguments within a function's parentheses. This is absolutely huge. If you can't remember all the different arguments a function takes, type the function and hit TAB within the parentheses. Try typing `read.table()` and his TAB while the cursor is inside the parentheses. Use the up and down arrows to scroll through the options, hit TAB again to insert one. Note that after you have used an argument it no longer appears in the list of options.

3.1.1.3 Ctrl + Shift + 1/2/3/4 to turn one of the panes to full-screen

Thanks to Diana for writing about practing that in her homework. Awesome feature that I've not used previously!

3.1.2 Workflow: projects

3.1.2.1 Disable saving of workspace for sure!

Let's all walk through this.

3.1.2.2 Another worthwhile preference for small-screens (like laptops)

Make sure that the RMarkdown preference is set to open in: **Window**. See Figure **??**.

3.1.2.3 Opening RStudio Projects from the OS (by clicking in the Finder)

- You can open an RStudio project by double clicking the RStudio Project icon from, for example, a Mac Finder window. It lives in a directory of the same name (but it has a `.Rproj` exension.)
- Or if you are a command line type, use, for example `open my_project.Rproj` from the Terminal.
- You can open as many RStudio projects as you like at a time.
- Each RStudio project launches its own, completely separate R session!
- Interestingly, if you click on the `.Rproj` file of a project that is open, RStudio will open another instance of that project. So, don't click on the `.Rproj` file for a project that is already open!
  - (In other applications on the Mac that will typically just take you to the currently open docuemnt, but not so with RStudio.)
- Use cmd-TAB to switch between open RStudio projects.

3.1.2.4 Opening RStudio Projects from RStudio

- When you open existing projects using the "File->Open Project…" menu option or with the "File -> Recent Projects" menu option and you currently have RStudio open "in another project,"" then the new project that you are opening jumps in "on top of" the previous one. It looks like your previous

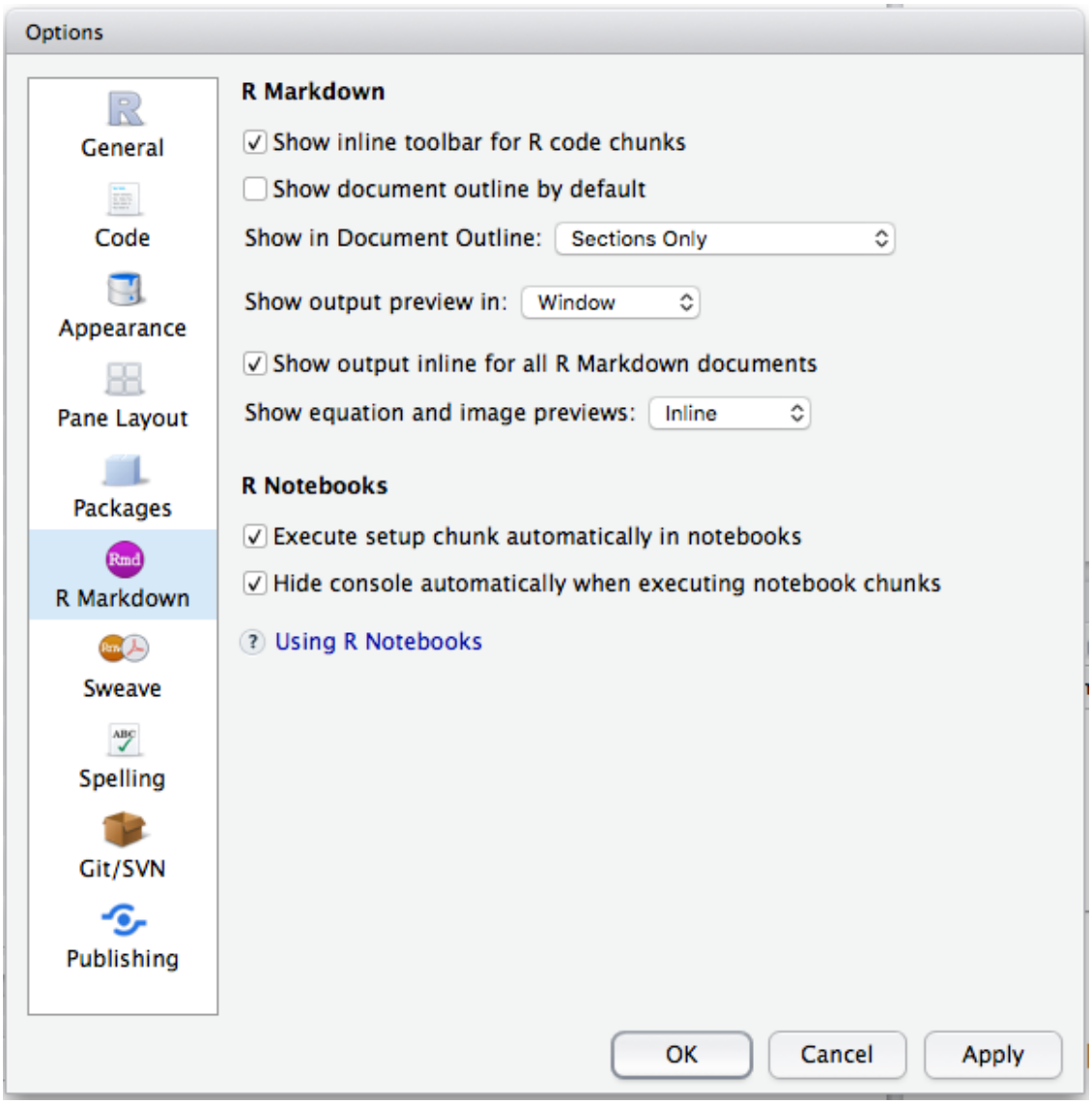


Figure 3.1: To read RMarkdown output in a separate page (highly recommended for laptops) choose "RMarkdown" on the left and choose "Window" from the dropdown menu, and click OK.

project has vanished into the ether. The OS thinks there is only one RStudio open, an it has the most recently opened project in it. WHERE'S MY OTHER ONE?!
- You can get back to it by clicking the project dropdown in the upper right of the project.
- However, if you switch between projects this way it restarts R each time you switch back to your project so it takes a lot of time and it is super-annoying.
- If you are working concurrently in multiple projects, I recommend opening them from the Finder (or Terminal) and switching between them using **Cmd-TAB**.

3.1.2.5 What is the `.Rproj` file, really

It is just a text file that stores some information and any project-specific preferences if there are any. Here is what `rep-res-eeb-2017.Rproj` looks like if you open it with a text editor:

```
Version: 1.0

RestoreWorkspace: Default
SaveWorkspace: Default
AlwaysSaveHistory: Default

EnableCodeIndexing: Yes
UseSpacesForTab: Yes
NumSpacesForTab: 2
Encoding: UTF-8

RnwWeave: knitr
LaTeX: pdfLaTeX

AutoAppendNewline: Yes
StripTrailingWhitespace: Yes

BuildType: Website
```

3.1.2.6 R in an RStudio project launches in the project directory

- This makes reproducibility much easier. You can find and load files using *relative* paths.

- Everything you might be accessing from R (data, scripts, etc.) or outputting from R will be easy to get to if they are "in the project"
- When we say that a file is "in the project" we mean that it is stored on disk somewhere within the project directory.
- The project directory (sometimes called the *root* of the project directory) is just the directory that contains the `.Rproj` file.
- Expert user tip: `rprojroot::find_rstudio_root_file()` (part of the `rprojroot` package) let's you find the root of an RStudio project directory. This can be helpful sometimes….

3.1.3 Workflow: scripts

- Script editor window vs console window
- Keyboard shortcuts for evaluating codes in your scripts:
  - **Cmd-Return** (sends current line to console and advances cursor to next line)
  - **Highlight with Cmd-Return** (send highlighted code to console)
    * For this, **Shift-up-arrow** and **Shift-down-arrow** are good for highlighting.

* As is **Shift-Command-right-arrow** or **Shift-command-left-arrow**.

3.2 Let's talk about the pipe %>%

For anyone who had ever worked comfortably in Unix for a long time, and was used to chaining the output of one utility in as the input for another utility using the pipe: `|`, R's syntax for composition of functions was always super cumbersome and required all sorts of nasty, nested parentheses.

Consider this simple set of operations: imagine we want to

1. simulate 1000 gamma random variables, G, with parameters $\alpha = 5$ and $\beta = 1$,
2. for each G simulate a Poisson random variable with mean (`lambda`) G.
3. take the `sqrt` of each such variable
4. compute the variance of the result

This can all be done in one line, but is ugly!

```
# set random seed for reproducibility
set.seed(5)

var(sqrt(rpois(n = 1000, lambda = rgamma(n = 1000, shape = 5, scale = 1))))
```

```
## [1] 0.5828768
```

It doesn't matter how stylishly you include spaces in your code, this is just Fugly!

You can write it on multiple lines, but it is friggin' ghastly! Maybe worse than before.

```
set.seed(5)

var(
  sqrt(
    rpois(n = 1000, lambda = rgamma(
      n = 1000, shape = 5, scale = 1
    )
    )
  )
)
```

```
## [1] 0.5828768
```

The problem is that the order in which the operations are done does not match the way things are written: the first thing to get done is the call to `rgamma`, which is nested deeply within the parentheses.

Enter the R "pipe" symbol. It is not as convenient to type as `|`, but you can make it quickly with the keyboard shortcut `cmd-shift-M`: `%>%`. This was introduced in the `magrittr` package, and the `tidyverse` imports the `%>%` symbol from `magrittr`.

Behold!

```
library(tidyverse)
set.seed(5)

rgamma(n = 1000, shape = 5, scale = 1) %>%
  rpois(n = 1000, lambda = .) %>%          # pass G is as the lambda parameter using the dot: .
  sqrt() %>%                               # no dot here, so the previous result is just the first argu
  var()                                    # same here
```

```
## [1] 0.5828768
```

That is a hell of a lot easier to read! It gives me goose bumps it is so elegant.

The `%>%` symbol says, "take the result that occurred before the `%>%` and pass it in as the `.` in whatever follows the `%>%`." Furthermore, if there is no `.` in the expression after the `%>%`, simply pass the result that occurred before the `%>%` in as the *first argument* in the function call that comes after the `%>%`.

This type of "chaining" of operations is particularly powerful when operating on `tibbles` using `dplyr`

3.3 Tibbles and "rectangular" data

- gonna talk a little about data types too.
- Chinook CWT data example.
- Get comfy with the `View()` function!

3.3.1 Tibble excercises

I'm gonna just blast through these here in case people are curious. These are answers I would use.

1. Use `class()`

```
class(mtcars)
```

```
## [1] "data.frame"
```

```
class(tibble::as_tibble(mtcars))
```

```
## [1] "tbl_df"     "tbl"        "data.frame"
```

Hey! does everyone see the `tibble::as_tibble()` there? The `::` is the "namespace addresser". It lets you run a function from a library without loading the library. If you already have done `library(tidyverse)` you would have loaded the `tibble` library and could just write `as_tibble(mtcars)` but I wanted to be explicit about where the `as_tibble()` function comes from. (As an aside, it turns out that this is how you would write it if you were writing code for a package.)

2. Let's do it first as a `data.frame`:

```
library(tidyverse)
df <- data.frame(abc = 1, xyz = "a")
df$x
```

```
## [1] a
## Levels: a
```

```
df[, "xyz"]
```

```
## [1] a
## Levels: a
```

```
df[, c("abc", "xyz")]
```

```
##   abc xyz
## 1   1   a
```

And then we can do it again as a `tibble`

```
df <- tibble(abc = 1, xyz = "a") %>%
  as_tibble()
df$x
```

```
## Warning: Unknown column 'x'
```

```
## NULL
```

```
df$xyz
```

```
## [1] "a"
```

```
df[, "xyz"]
```

```
## # A tibble: 1 × 1
##     xyz
##   <chr>
## 1     a
```

```
df[, c("abc", "xyz")]
```

```
## # A tibble: 1 × 2
##     abc   xyz
##   <dbl> <chr>
## 1     1     a
```

Aha! Things to notice are:

a. `data.frame()` coerces to factors.
b. tibble doesn't do partial name matching `$x`≠`$xyz`
c. Square bracket extraction of a single column of a tibble retains its tibbleness. Not so with `data.frame`. With `data.frame` it gets turned into a vector.
d. `$` extraction with `tibble` returns a vector.

3. Let's make `mtcars` a `tibble`

```
mtcars_t <- as_tibble(mtcars)
var <- "mpg"

# this will get the "mpg" column out but retain it as a tibble
mtcars_t[, var]
```

```
## # A tibble: 32 × 1
##      mpg
##    <dbl>
## 1   21.0
## 2   21.0
## 3   22.8
## 4   21.4
## 5   18.7
## 6   18.1
## 7   14.3
## 8   24.4
## 9   22.8
## 10  19.2
## # ... with 22 more rows
```

```
# and this will just grab the column as a vector
mtcars_t[[var]]
```

```
##  [1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2
## [15] 10.4 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4
## [29] 15.8 19.7 15.0 21.4
```

4. Remember that non-syntactic names (those that do not start with a letter or underscore and which include characters other than - and "_" and ".") must be enclosed in backticks. Let's get our data:

```
annoying <- tibble(
`1` = 1:10,
`2` = `1` * 2 + rnorm(length(`1`))
)
```

OK, now let's do the questions:

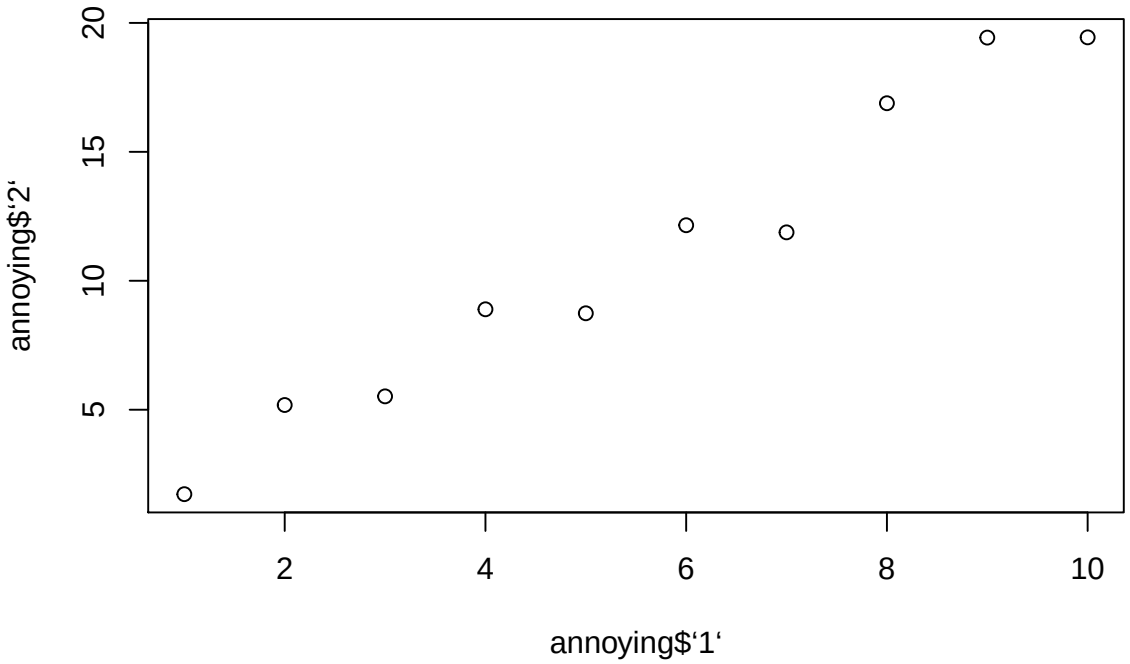
1. Use dollar sign with backticks:

   ```
   annoying$`1`
   ```

   ```
   ##  [1]  1  2  3  4  5  6  7  8  9 10
   ```

2. Use dollar sign with backticks

   ```
   plot(annoying$`1`, annoying$`2`)
   ```



3. Use `mutate` (we haven't talked about this yet) with backticks

   ```
   annoying %>%
     mutate(`3` = `2` / `1`)
   ```

   ```
   ## # A tibble: 10 × 3
   ##      `1`       `2`      `3`
   ##    <int>     <dbl>    <dbl>
   ## 1      1  1.725450 1.725450
   ## 2      2  5.182598 2.591299
   ## 3      3  5.517888 1.839296
   ## 4      4  8.894382 2.223596
   ## 5      5  8.740021 1.748004
   ## 6      6 12.153600 2.025600
   ## 7      7 11.878134 1.696876
   ## 8      8 16.885478 2.110685
   ## 9      9 19.429848 2.158872
   ## 10    10 19.439987 1.943999
   ```

4. Use `rename` (we haven't talked about this yet) with backticks

```
annoying %>%
  mutate(`3` = `2` / `1`) %>%
  rename(one = `1`,
         two = `2`,
         three = `3`)
```

```
## # A tibble: 10 × 3
##      one       two    three
##    <int>     <dbl>    <dbl>
## 1      1  1.725450 1.725450
## 2      2  5.182598 2.591299
## 3      3  5.517888 1.839296
## 4      4  8.894382 2.223596
## 5      5  8.740021 1.748004
## 6      6 12.153600 2.025600
## 7      7 11.878134 1.696876
## 8      8 16.885478 2.110685
## 9      9 19.429848 2.158872
## 10    10 19.439987 1.943999
```

5. Look it up with `?enframe`. It turns out that `enframe()` is super useful.
   Often you will have a vector of values with names associated with it. For example:

   ```
   v <- c(1, 3, 4, 10)
   names(v) <- c("a", "b", "b", "c")
   v
   ```

   ```
   ##  a  b  b  c
   ##  1  3  4 10
   ```

   If you want to deal with this type of vector in the tidyverse, you can enframe it into a tibble:

   ```
   enframe(v)
   ```

   ```
   ## # A tibble: 4 × 2
   ##    name value
   ##   <chr> <dbl>
   ## 1     a     1
   ## 2     b     3
   ## 3     b     4
   ## 4     c    10
   ```

   By default it makes columns of "name" and "value". That is awesome!
6. Do `package?tibble` and read through it to find that the answer we want is `tibble.max_extra_cols`.

3.4 Data import

- Why use `readr` instead of the base-R reading functions? Plenty of reasons.
- Explicit column specifications if you want to do that.

3.4.1 RStudio's GUI importer

- This is a great way to start importing CSV files.
- Don't do it every time! use the code that it creates to make reading your data in reproducible.

Chapter 4

Week Three Meeting

4.1 Git Basics

Goals for Today:

- Explain what git is (and how it is different than GitHub)
- Introduce the sha-1 hash (for fun!)
- Get familiar with RStudio's very convenient interface to git
  - staging files
  - unstaging file
  - viewing differences between staged and unstaged files
  - committing files
  - viewing the commit history

4.1.1 An overview of Version Control Systems (VCS)

- Git is a type of VCS
- At its crudest, a VCS is a system that provides a way of saving and restoring earlier versions of a file.

4.1.1.1 A typical VCS for a non-computer programmer

- Start writing `my_manuscript.doc`.
- At some point worry that MS Word is going to eat your file, so,
  - Make a "backup" called `my_manuscript_A.doc`
- Then, before overhauling the discussion, save the current file as `my_manuscript_B.doc`.
- Email it to your coauthors and then have a series of files with other extensions such as the initials of their names when they edit them and send them back.
- Etc.
- Disadvantages:
  - Hard to find a good record of what is in each version. (Wait! I liked the introduction I wrote three weeks ago…where is that now?)
  - A terrible system if you have multiple files that are dependent on one another (for example, figures in your document, or scripts and data sets if you have a programming project.)
  - If you decide that you want to merge the changes you made to the discussion in version `_C` with the edits on the introduction in version `_K`, it is hard.

4.1.1.2 Other popular VCS systems

- rcs, cvs, subversion, etc.
- These all had a "Centralized" model:
  - You set up a repository on a server that has the full version history,
  - then each person working on it gets a copy of the current version, and nothing more.
  - They can submit changes back to the central repository which tries to deal with conflicting submissions.
  - You need to be online to do most operations.
- I used a few of these, and missed them for a few weeks when I switched to Git, but then never looked back and couldn't imagine using them again.

4.1.1.3 The Git model — Distributed Version Control

- Git stores "snapshots" of your collection of files in a repository
- For our work, the "collection of files" will be "the stuff in your RStudio project"
  - Another reason it is nice to keep everything you need for a project together in a "project directory"
  - Though git scans your project directory for new additions and changes, it will not add a new file, or add new changes, to the repository until you *stage* and subsequently *commit* the file.
- When you clone a repository, **you** get the whole version history
- When someone else clones that repository, **they also** get the whole version history.
- Git has well-developed features for merging changes made in different repositories
  - But, for today, we will talk mostly of a single user interacting with git.

4.1.2 Git versus GitHub

- They are not the same thing!
- Git is software that you can run on your own machine for doing version control on a repository.
  - It can be *entirely* local. i.e. only on your hard drive and nowhere else.
  - This is super-useful for any project, because solid version control is great to have.
- GitHub is a website, with tools powered by Git (and many that they brewed up themselves) that makes it very, very easy to share git repositories with people all over the place.

4.1.2.1 Is everything on GitHub public?

- No! Many companies use GitHub to host their proprietary code
  - They just have to pay for that…
- By default, you can put anything on GitHub for free as long as it is under a fairly free-use (open source type) license and it is available to anyone
- If you want a private repository, as an academic affiliate you just have to ask and GitHub will give you unlimited private repos for free.
- And if you are a student and you have not yet done so, go to https://education.github.com/pack to sign up for your free student pack.

4.1.3 Using git through RStudio

- Now we can do a few things together to see how this works.
- Most of the action is in the Git Pane…
- Today we will talk about:
  - Staging files (preparing them to be **committed**)
  - Committing files (putting them into the repository)
  - viewing differences between staged and unstaged files

  - committing files
  - viewing the commit history

4.1.3.1 Two final configurations before starting:

- Open the shell (Tools->Shell…) and issue these two commands, replacing the name "John Doe" with yours, and his email with yours.
  - You may as well use the email address that you gave to GitHub, though it doesn't necessarily have to be

    ```
    git config --global user.name "John Doe"
    git config --global user.email johndoe@example.com
    ```

    You only need to do this once.
- Finally, if you are using a Mac, configure it to cache your GitHub credentials so you needn't give your password every time you push to it:

  ```
  git config --global credential.helper osxkeychain
  ```

4.1.3.2 The status/staging panel

- RStudio keeps git constantly scanning the project directory to find any files that have changed or which are new.
- By clicking a file's little "check-box" you can stage it.

- Some symbols:
  - **Blue-M**: a file that is already under version control that has been modified.
  - **Yellow-?**: a file that is not under version control (yet…)
  - **Green-A**: a file that was not under version control, but which has been staged to be committed.
  - **Red-D**: a file under version control has been deleted. To make it really disappear, you have to stage its disappearance and commit. Note that it still lives on, but you have to dig back into your history to find it.
  - **Purple-R** a file that was renamed. (Note that git in Rstudio seems to be figuring this out on its own.)

4.1.3.3 The Diff window

- Shows what has changed between the last committed version of a file and its current state.
- Holy smokes this is convenient
- (Note: all this output is available from the command line, but the Rstudio interface is very nice, IMHO)

4.1.3.4 Making a Commit

- Super easy:
  - After staging the files you want to commit…
  - Write a brief message (first line short, then as much after that as you want) and hit the commit button.

4.1.3.5 The History window

- Easy inspection of past commits.
- See what changes were made at each commit.

4.1.4 Go for it everyone!

- Make some changes and commit them yourselves.

- Add some new files to the project, and commit those.
- Get familiar with the diff window.
- Check the history after a few commits.

4.1.5 How does git store and keep track of things

- Everything is stored in the .git folder inside the RStudio project.
- The "working copy" gets checkout out of there
- Committed changes are recorded to the directory

4.1.5.1 What is inside of the .git directory?

We can use R to list the files. My `rep-res-course` repository that hold all the materials for a course like this one looks like:

```
## check out this file-system command in R
dir(path = ".git", all.files = TRUE, recursive = TRUE)
```

The output from that command looks something like this:

```
  [1] "#MERGE_MSG#"                                   "COMMIT_EDITMSG"
  [3] "COMMIT_EDITMSG~"                               "config"
  [5] "description"                                   "FETCH_HEAD"
  [7] "HEAD"                                          "hooks/applypatch-msg.sample"
  [9] "hooks/commit-msg.sample"                       "hooks/post-update.sample"
 [11] "hooks/pre-applypatch.sample"                   "hooks/pre-commit.sample"
 [13] "hooks/pre-push.sample"                         "hooks/pre-rebase.sample"
 [15] "hooks/prepare-commit-msg.sample"               "hooks/update.sample"
 [17] "index"                                         "info/exclude"
 [19] "logs/HEAD"                                     "logs/refs/heads/gh-pages"
 [21] "logs/refs/heads/master"                      "logs/refs/remotes/origin/gh-pages"
 [23] "logs/refs/remotes/origin/master"           "objects/00/906f99e192ff64b4e9e9a0e5745b0a4f841cbd"
 [25] "objects/01/ab18d4ce04fb06532bb06ed579218fef89d478" "objects/02/74554e0b574b9beb2144f26ad392583005
 [27] "objects/03/2d224bf78798e8b9765af6d8768ade14694a9d" "objects/04/03d552ab37b0bcaeebed0ac3068d669261
 [29] "objects/04/4a12f8ccc12a4a5ba84ab2bf5a1ae751feea6f" "objects/04/9ec3065bb0434ded671fa83af5ade803bc
 [31] "objects/04/ea8efb1367727b081dea87e63818be0a4d02f0" "objects/05/b22ecc373d5058e36d7ca773a4475c46da
 [33] "objects/07/8831b46c9b63e8c2d50b79304ed05de9274c28" "objects/07/b57af2a0cbd0545a6cd3e93f10cc5d768e
 [35] "objects/08/674e6e4d534b3424e2629510d20bb6d1b0be94" "objects/08/8b282d5b978dc1ff6eef3871d3fb3a9256
 [37] "objects/09/565dcd10d7adc0551783b443e8fd71486b3997" "objects/09/6828a0cfb96f30d6e99cfc04a5c1686b9e
 [39] "objects/0a/30fe678abc342c58daab0ad42163b371babda0" "objects/0a/b71109dae6e5711755feddfb06b81b1376
 [41] "objects/0b/442fdfc183783537985c17151ae3483fa00cf6" "objects/0b/c0451fd0e7081a7db05fdd38b12870bdca
 [43] "objects/0c/0f7cf8c73d901795dad4bd5f504c53c3bf2093" "objects/0d/14a7a2a19ffc3b9820f011e3270c965a5f
 [45] "objects/0e/35cfb4d55e52d27083b8d2eccab9296b920d76" "objects/0e/7bba5882077a8b00a76d3eabe6b23cadc6
 [47] "objects/0e/8abf4cc0885a727ee2459fdbb272828e267cc4" "objects/0f/1f3f7be7787d5d44dc1155f3b7a44eddc9
 [49] "objects/10/54d2e7a9baf61618521c522b15db40855b3431" "objects/11/7d874e1616500b5fba51b9f0ee1e8d0fbe
 [51] "objects/11/c33cc1d5c8de7c7cbf7257b7d32f7ca3d458ef" "objects/14/1ccea514e106e20eef47b791a23e036d1f
 [53] "objects/15/cc3a6f15dadb3446ad0af34a3ecde8d81d65f9" "objects/15/ddaf45bf00c3ef2d8f499ebd6dc3a86bf9
 [55] "objects/16/0c9386dfa9707d81fbbbcc52f0c7638703f9a9" "objects/16/8ee93b6a4612dbd76bc06a49460df9f9f6
```

Yikes!

4.1.5.2 How does git know a file has changed?

- Does it just look at the modification date?
- NO! It "fingerprints" every file, so it knows when it has changed from the most recent committed version.
  - Demonstration. Change a file. Save, then undo the change and save again…Git knows the file has been changed back to its "former self"
- SHA-1 hashes. We will learn more about those later.

- You will see things like `ed00c10ae6cf7bcc35d335d2edad7e71bc0f6770` all over in Git-land.
- You can treat them as very specific names for different commits.

4.1.5.3 What should I keep under version control?

- General rule: don't keep derived products.
  - i.e. If you have an Rmd file that creates an html file, there isn't much need to put the html file under version control with git, because you can just regenerate it by Knitting the Rmd file.
- Do keep data, source code, etc.
- Sometimes certain outputs and intermediate results from long calculations can be committed so that you don't have to run a 4 hour analysis to start where you were before.
- For such results, consider `saveRDS()` with the `compress = "xz"` option (and its companion `readRDS()`.

4.1.5.4 How can I make git ignore certain files?

- The `.gitignore` file!
- File names (and patterns) in the `.gitignore` file are ignored *recursively* (down into subdirectories), by default.
- Files won't be ignored if they are already in the repository.
- Example: `*.html`

4.2 Pushing and Pulling With GitHub

4.2.1 Creating a Repository on GitHub and the initial push

When you have an RStudio project under git version control on your laptop or desktop computer, creating a remote repository on GitHub is quite easy. A few steps:

1. Upper right corner: "create new" button (a "+" with a little triangle.) Choose "New Repository"
2. Give it a name. It makes most sense to name it the same as the RStudio project you want to push up there. So, for example, if my project file was `boing.Rproj`, I would name the repository `boing`.
3. Add a 5 or 6 word description if you want.
4. Choose **public** or **private**
5. DO **NOT** choose to "Initialize this repository with a README". You likely already have a README. Initializing the repository with one will create headaches.
6. Also, don't add a `.gitignore` or a license (select "none", which should be the default, for both of those)
7. Click the green "Create Repository" button

That will take you to another screen. In the middle find the code box below the heading, **…or push an existing repository from the command line**.

1. Copy that two lines of code from you web browser. It will look something like this:

```
git remote add origin https://github.com/eriqande/boing.git
git push -u origin master
```

but it will be specific to the repository you just made, so the URL and name of the repo will be different than what you see above. Note, you can copy the lines by clicking the "copy this text" icon on the right side of the page.

2. Go to RStudio, in the project that you want to push to GitHub, and choose "Tools-Shell". That will give you a terminal window. Paste the commands you copied into that terminal window and hit return.
3. It might ask you for your GitHub username and password.

Voila!

4.2.2 Subsequent pushes

Once you have pushed the repo up there. Try making some changes on your laptop, committing them, and then hitting the "Push" button on the git panel…

4.2.3 Assign collaborators

From the repository page on GitHub, choose "Settings" (on the upper right) and find the "collaborators" link (on the left).

If you have a private repository, you can add GitHub user `eriqande` (that's me…) to it and I will be able to view it and give comments and suggestions.

4.3 Next Week's Assignment

- With luck, we will get everyone's projects up to GitHub before the end of our session today. However, if we don't, please get that done ASAP.
- The big assignment is to read the Data Transformation chapter in "R for Data Science." *Warning*: This is a long and meaty chapter, so get an early start! The chapter goes through what you need to know to leverage all the `dplyr` goodness. Please work all the examples, and do the exercises. In fact, when you are working through the examples with the `nycflights` data set, you should, after each example, try to do the same type of operation on your own data set.

Chapter 5

Week Four Meeting

This was a bit of a free-form discussion on a variety of topics.

5.1 Knit your README.Rmd files

First thing we talked about was the fact that GitHub will render a `README.md` file to an html web page that is nice and easy to read. It will sort of render a README.Rmd file, but it won't do everything to it. Namely:

1. The YAML header block comes out as a table.
2. It will **not** evaluate all the R code and deliver the results.

Rather, it is necessary to locally *knit* the `README.Rmd` file to create a `README.md`, then this `README.md` file must be committed to the repository and pushed. This needs to happen each time you have updated the `README.Rmd` file.

Note that your `README.Rmd` file should start with the following:

```
cat(readLines("inputs/readme-header.txt"), sep = '\n')
```

````
---
title: "NameOfPackage"
date: "`r format(Sys.time(), '%d %B, %Y')`"
output:
  github_document:
    toc: true
---

<!-- README.md is generated from README.Rmd. Please edit that file -->

```{r, echo = FALSE}
knitr::opts_chunk$set(
  collapse = TRUE,
  comment = "#>",
  fig.path = "readme-figs/"
)
```

Then start adding your text here...
````

5.2 Changing to factors

Mikki had a question: she wanted to have a column that contained 1's and 2's as factors. her data set had several entries that were "1,2". She wanted to convert those to 2's and then make them all factors. We discussed how this could be done with dplyr. The important message was that dplyr does not change the original input variable, but in the output, you can "mutate over the top of an existing variable".
(i.e. in the output the column will have been changed, but not in the original input data frame.)

5.3 The `group_by()` function

We spent a bit of time going over how to think about what the `group_by()` function does. Eric likes to think of it as breaking your original tibble up into a lot of different tibbles, according to the grouping variables, after which, each little tibble gets sent to the following verb (`summarise()`, `mutate()`, `filter()`, etc.)

We talked about that fact that while it is quite natural to think about using the `group_by()` function in conjunction with `summarise()`, it is also very powerful to be able to use it in conjuntion with `mutate()`.

When you do a summarise, only the grouping variables and the newly-created summary variables get returned in the output tibble, and the rows are arranged by the grouping variables. When you do a `group_by()` and then `mutate()` all of the columns get returned and there is no automatic arranging that goes on.

Chapter 6

Week Five Meeting

6.1 Some things regarding people's repositories

6.1.1 Data Compression

We have been endorsing `.csv` as a good format for data, and *it is*, because it is human-readable and easily parsed into tibbles. However, when you have very long tibbles, it is not necessarily the most space-efficient format. Large `.csv` files can take up much more space on your hard drive *than they should*.

What do we mean by "*than they should?*" in that context? This has to do with how much *information* is in the file, where information is used in the context of *information theory*. Often tibbles will have columns that have fairly "redundant" information—for example, in a mutlispecies salmon data set, one column might have entries that are either "Chinook", "coho", or "steelhead". It takes a few bytes to store each one of those words, and if they are used in a column that has millions of rows, that can add up to a lot of space on your hard drive. Colloquially, *data compression* is the art of finding ways of using short "code-names" for things or patterns that occur frequently in a file, and in so doing, reducing the overall file size.

The consequences of data compression can be profound and wonderful. You can reduce the size of a file, sometimes by an order or magnitude or more. There are a few good choices available for compressing your data (making it smaller.). Note that doing so often makes it a little harder to edit your data set; however, if your data set is not going to change, and it is large, then it makes sense to compress it—especially if it is so big you would rather not (or can't) put it on GitHub.

6.1.1.1 gzip

If you are working on a Mac, you have the Unix utility `gzip`. We will illustrate its use on Katie's big salmon data set, `ASL_merged.csv`. Let's see how big that is. We can use the Unix utility `du` (stands for "disk usage").

```
# give this command on the Terminal in the directory where the file lives:
du -h ASL_merged.csv
```

The results comes back:

```
322M    ASL_merged.csv
```

Whoa! This file is 332 Megabytes. That is quite large!

However, we can compress it like this:

```
gzip ASL_merged.csv
```

When we do that, it compresses the file and renames it to have a `.gz` extension on it: `ASL_merged.csv.gz`. We can then see how big that file is:

```
du -h ASL_merged.csv.gz
```

tells us:

```
11M ASL_merged.csv.gz
```

Whoa! We went from 332 Megabytes to 11. It is just 3% of its original size (and small enough that you can safely put it on GitHub).

One very nice feature is that gzipped files can be read in directly by the functions of the `readr` package. So, for example, `read_csv()` works just fine on the gzipped version of Katie's massive salmon data set:

```
# this works the same as it would on the ungzipped file
salmon <- read_csv("data/ASL_merged.csv.gz")
```

6.1.1.2 xz comression with `saveRDS()`

Another method that can be even more efficient with tibbles is to store them as R objects using the `saveRDS()` function with the `xz` compression option. This has the nice advantage that all the data types of the variables (for example, if you had made factors out of some) will be preserved *exactly* as they are in the tibble when you save it.

Let's imagine we have read the tibble into the variable `salmon`, and all the column types were as we wanted. Then, we could save that tibble directly to a compressed file like this:

```
saveRDS(salmon, file = "ASL_xz.rds", compress = "xz")
```

Note that `compress = "xz"` option. Let's see how that did using `du` on the Unix terminal:

```
du -h ASL_xz.rds
```

tells us:

```
3.7M    ASL_xz.rds
```

Holy Smokes! Only 3.7 Megabytes. That is only 1.1% of its original size. Lovely!

In order to read that tibble back into a variable (named `my_var`, say) in R, you would use `readRDS()` like this:

```
my_var <- readRDS(file = "ASL_xz.rds")
```

Voila!