R Markdown for Psychology Graduate Students

Holly Zaharchuk

2020-05-04

Contents

Chapter 1

Welcome

I designed this guide to be a resource for psychology graduate students looking to streamline their research pipelines. With R Markdown, you can load, clean, manipulate, analyze, and present your data in one environment. This guide focuses on the presentation piece, with information on creating slides, posters, manuscripts, CVs, and reports in several formats, including HTML, PDF, and Microsoft Word.

1.1 Background

This guide assumes a basic level of familiarity with R and RStudio. If you don't know how to use either of these, there are several beginner tutorials you should check out first. I have also created primers for Beginner and Intermediate R in Chapter 9.

1.2 Getting started

What is R Markdown?

Markdown is a specific markup language with plain text-formatting syntax. R Markdown is a specific markdown variety.

R Markdown and R are not the same thing. R Markdown combines R code (or code from other programming languages) and markdown in the RStudio integrated development environment (IDE). This allows you to embed code and text in the same document.

You can install the `rmarkdown` package from CRAN or GitHub:

```
install.packages("rmarkdown")
# or the development version:
# devtools::install_github("rstudio/rmarkdown")
```

1.2.1 Tips

- Treat your data as read-only
- Comment code early and often
- Keep code chunks small
- Label chunks to help with diagnosing issues
- Nest all files under one directory (if possible)

1.2.2 General reference documents

- R Markdown Guide
- R Markdown Cheat Sheet
- R Markdown Reference Guide
- Keyboard shortcuts
- `knitr` documentation

Chapter 2

Parts of a document

1. YAML header
2. Markdown
3. Code chunks

2.1 YAML header

The first part of your document is called the YAML header. This is where you set the global options for the output and formatting. The YAML header appears at the top of your document, and is defined by three dashes/hyphens at the beginning and end.

In the example below, I show the YAML header for a set of `revealjs` slides. I've included basic information, like the title and date, in addition to template-specific parameters, like whether there should be slide numbers or not. Section 6.2 has more information on setting YAML formatting parameters.

```yaml
---
title: "Cognitive Brownbag"
author: "Holly Zaharchuk"
date: "February 19, 2020"
output:
  revealjs::revealjs_presentation:
    theme: white
    center: true
    transition: slide
    self_contained: true
    reveal_options:
      slideNumber: true
      progress: true
---
```

2.2 Markdown

The plain text-formatting syntax of R Markdown allows for conversion to multiple document types. The image below shows an example of the basic syntax. The # denotes a header, while the numbered list behaves like you would expect one to in Word. However, unlike Word, the actual numbers don't matter; I could've put all 1's here, and R Markdown would've formatted them for me. Go to Chapter 5 for more information on markdown syntax.

```
# Parts of a Markdown document

1. YAML header
2. Markdown language
3. Code chunks
```

2.3 Code chunks

Code chunks are one of the core features of R Markdown. Code chunks are set apart from markdown by three backticks at the beginning and end. In curly brackets after the first set of backticks, you specify the coding language you want to use (here, it's R, with a lowercase r).

You can also add other arguments, like a name for the chunk (here, it's `setup`), and specific chunk options. The example I've provided below is the first chunk in my R Markdown document. It establishes the default chunk options with `knitr::opts_chunk$set`. You can see that while I've set `echo = TRUE` globally, so that all of my code chunks appear in the document by default, I set `echo = FALSE` for this specific chunk. A full list of chunk options can be found here.

```
# This is a chunk of R code that adds an image
knitr::include_graphics("images/example_chunk.png")
```

````
```{r setup, echo = FALSE}
## R setup ##

chooseCRANmirror(graphics = FALSE, ind = 1)
knitr::opts_chunk$set(warning = FALSE, message = FALSE, echo = TRUE, eval = TRUE)

## Load packages ##

# Package list
pkg_list <- c("plyr","tidyverse", "data.table", "ggplot2", "kableExtra")

# Load packages
pacman::p_load(pkg_list, character.only = TRUE)
```
````

There are multiple ways to run code chunks to test them in RStudio before creating your output. You can run code like you would in R by highlighting the relevant lines of code and hitting CTRL/command + enter. You can also hit the green "play" button in the upper right-hand corner of the chunk.

Each chunk is an island, so if you haven't run a previous chunk that contains some variable you need in the chunk you want to run, it'll throw an error. At the top right of your open .Rmd document in RStudio, you'll also see a **Run** dropdown menu. There, you can choose different options for running certain code chunks.

Tip: you can use the chunk option "cache = TRUE" for very time-consuming chunks, but there are some catches as described here.

Chapter 3

Outputs

R Markdown can transform plain text and code into several different document formats. There are also multiple ways to **Knit** or `render` the output.

3.1 Output options

3.1.1 html_document

HTML is overall the most flexible. It supports the types of content we're interested in creating as graduate students—tables, graphs, and the like—and you can easily transform your HTML output to a PDF with `pagedown::chrome_print(file)` if you have Google Chrome. The poster template I use follows this process.

3.1.2 pdf_document

If you're familiar with LaTeX, you may be inclined to output directly to a PDF, since you can include inline LaTeX code in your documents (more on this in Chapter 6). The CV and manuscript templates that I use rely on the fine-grained typsetting capabilities of LaTeX. To create a PDF, you need to have LaTeX installed locally. If you don't already, you can install a `tinytex` distribution through the R console.

```
install.packages("tinytex")
```

3.1.3 word_document

You can also output to Microsoft Word and Powerpoint. I often work with colleagues who prefer to edit in Word, so sometimes I need to do this. I prefer not to if I can help it though, because you lose several important functions. For

example, chunk options for aligning and setting the size of graphs and other images don't work, and `kable` tables tend not to appear correctly.

3.2 Rendering options

3.2.1 Knit

You can specify a particular output type in the YAML header with the **output** option. Then, you can press the **Knit** button or CTRL/command + shift + K to create or "knit" the document.

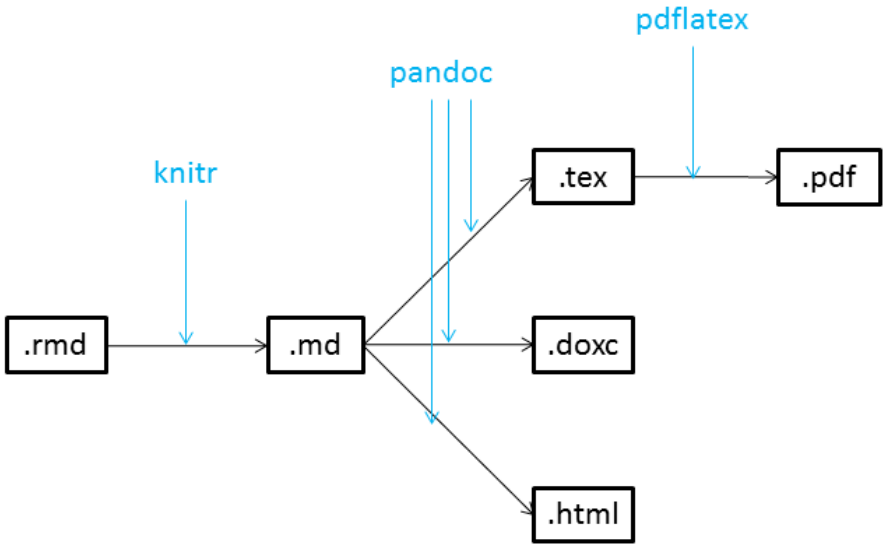
Be sure to pay attention to indentation and colon placement in your YAML header, especially as you start to specify options for particular output types. As a rule of thumb, if you have anything more than **output: output_format**, you need to indent every new line, and have a colon at the end of the previous line. See Section 4.3 for an example of this.

3.2.2 `rmarkdown::render`

Instead of knitting, you can also render files in the console with `rmarkdown::render(file, output_format)`. This is essentially what **Knit** is doing. This approach allows you to create multiple output types quickly and easily. I'm usually customizing my output very specifically for one output type by adding inline CSS or LaTeX code, so I really only use `render` when I'm making parameterized reports.

3.3 Rendering process

When you **Knit** or `render` your document, there's a particular sequence of events that happens under the hood. Your R Markdown document is piped through to pandoc by the `knitr` package, which runs your code chunks and knits them together with the plain text you've included. Pandoc ultimately handles the conversion to a particular output format. Understanding this process can help you troubleshoot when you run into issues. You can find more on troubleshooting in Chapter 8.


pdflatex
pandoc
knitr
.tex
.pdf
.rmd
.md
.doxc
.html

Chapter 4

Templates

1. Built-in templates
2. Templates from R packages
3. User-defined templates

4.1 Built-in templates

R Markdown has several output templates built in, and you just have to specify them with the YAML **output** parameter. These include:

- Presentations
  - ioslides and Slidy for HTML
  - Beamer for PDF
- Interactive Shiny documents and presentations

4.2 Templates from R packages

You can also download specific templates from CRAN and GitHub. You use them the same way as the built-in templates by specifying them in the YAML header. A few that I've used or played with include:

- Presentations: `revealjs`
- CVs: `vitae`
- Academic posters: `posterdown`
- APA articles: `papaja`
- Journal templates: `rticles`
- HTML theme: `prettydoc`

4.3 User-defined templates

There are also ways to include templates for other output types in the YAML header.

4.3.1 PDF

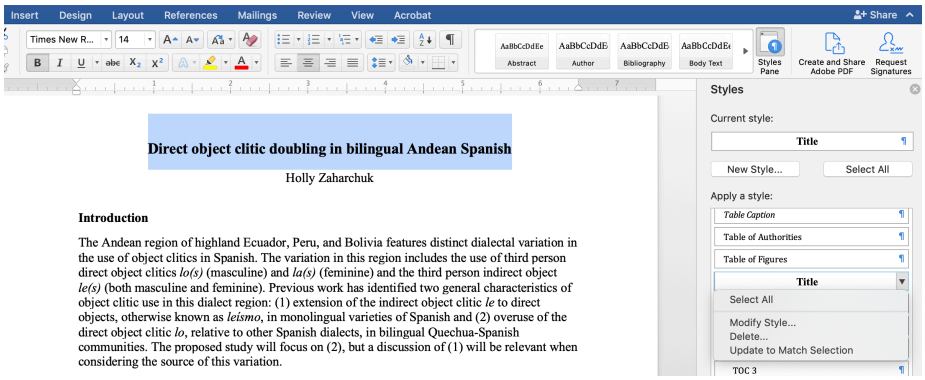
For PDFs, you can include LaTeX templates, but you need to pay attention to the use of $. The PLOS template shown in the image below failed to compile when I first downloaded it, because $ are special characters for pandoc. After I added a second $ throughout the document (you just use find and replace for this), it worked. See Section 6.4.1 for calling specific LaTeX packages in R Markdown.

```
---
output:
  pdf_document:
    template: plos_latex_template.tex
---
```

4.3.2 Word

You can also set up Word templates. To do this, you need to use the Styles Pane in the Word Document you want to use as a template. Highlight the text you want to format, make the desired changes, then find the style that applies to that section, and find "Update to Match Selection" in the dropdown menu.

In the example below, you can see that I'm editing the "Title" format, so I need to update that particular entry in the Styles Pane. You have to do this for every text element in the document, but once you've set it up once this way, you can simply include it the same way as the LaTeX template shown above.



Chapter 5

Content

The content of an R Markdown document includes the markdown text itself, as well as output from code chunks. Code chunks can output data, graphs, tables, and images. You can also reference variables from code chunks in markdown text.

5.1 Markdown overview

R Markdown: The Definitive Guide and this R Markdown cheatsheet provide comprehensive information on the typesetting capabilities of R Markdown. In general, R Markdown typesetting options include `*italics*`, `**bold**`, and `~~strike-through~~`. These are achieved by wrapping text in a certain number of asterisks or tildes. There are also `(parentheses)`, `[square brackets]`, and `"quotation marks"` that can have special functions in markdown, like creating hyperlinks: `[text](link)`.

With many of these typesetting characters, if you highlight the text you want to format (by clicking and dragging your cursor), you can just hit the character once to wrap the text automatically. This way, you don't have to go to the beginning and end of the text and place the characters individually.

Another note about R Markdown is that line spacing matters. For example, if I wanted to include bullet points after this sentence, they wouldn't render properly if I didn't hit enter twice before starting them. In other words, I need to have a full line of white space before bullet points and numbered lists. If you're having issues with your document rendering correctly, make sure you have line breaks between lists, paragraphs, and headers.

5.2 Special characters

If you want any special characters in R Markdown, LaTeX, or pandoc to appear as text, rather than having them perform some function, you need to "escape" them with a backslash. For example, pound signs/hashtags, backslashes, and dollar signs need to be preceded by a backslash.

This also applies to any chunk outputs that contain strings with special characters, as with `knitr::kable` tables with LaTeX functions or characters (e.g., Greek letters like η to report partial eta-squared or functions like `\textit{p}` to italicize the text). Sometimes you even need multiple backslashes, so you may have to play around to troubleshoot if they're not rendering correctly. These kinds of rendering issues won't generally throw errors, so you'll have to check the output in the knitted document to make sure it looks the way you want.

Speaking of LaTeX, you can engage "math mode" by putting dollar signs around LaTeX math commands. This way, you can include fractions and binomials, math symbols, International Phonetic Alphabet (IPA) symbols, and the like in R Markdown (even if you're not outputting to a PDF). For example, I can write $e = mc^2$ in a sentence like this just by wrapping the equation in a single set of dollar signs, or I can use two sets to center the equation:

$$e = mc^2$$

5.3 Chunk output

Depending on the kind of content you're creating with R Markdown, whether it's a poster, manuscript, or internal lab document, there are several ways you can take code chunks and turn them into content.

5.3.1 Data

When I'm working on a project and checking in with my advisor on my progress, I display my raw data and analyses in R Markdown. My usual work-flow includes data pre-processing in MATLAB for EEG data and R for the ERP analyses and behavioral data. I get everything clean and ready to go in these scripts.

As I describe in Section 7.2, I keep separate scripts for each piece of the data analysis. I `source` them into one another, with a *global* script at the base with any general variables (like file names, HEX color codes for graphs, etc.) and custom functions. Once I've built out this processing pipeline with R scripts, that's when I'll `source` them into my R Markdown documents for statistical analysis and presentation. Once you've got your data loaded into R Markdown, you just use R code to run analyses and output them in your document as you would in a regular R script.

If I'm running regressions with `lm` from the `stats` package for example, I'll wrap the `summary` function around the output. I tend to save this as a variable, since

I typically want to grab individual values from the variable later (e.g., p values). You can put the new variable name on its own line or `print` it if you prefer; otherwise, you can just have a line with `summary(model)`, and it'll output the table in your document.

```
# Show first five rows of mtcars dataset
head(mtcars, 5)
```

```
##                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
```

```
# Provide summary statistics for miles per gallon (mpg) and weight (wt)
  # select is from dplyr
  # describe is from the psych package
mtcars %>% select(mpg, wt) %>% describe()
```

```
##     vars  n  mean   sd median trimmed  mad   min   max range skew kurtosis   se
## mpg    1 32 20.09 6.03  19.20   19.70 5.41 10.40 33.90 23.50 0.61    -0.37 1.07
## wt     2 32  3.22 0.98   3.33    3.15 0.77  1.51  5.42  3.91 0.42    -0.02 0.17
```

```
# Are car weight and miles per gallon correlated?
mpg_model <- lm(mpg ~ wt, mtcars)

# Save summary of model
mpg_summary <- summary(mpg_model)

# Output results
  # I could have put summary(mpg_model) or print(mpg_summary) instead if I preferred
mpg_summary
```

```
##
## Call:
## lm(formula = mpg ~ wt, data = mtcars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -4.5432 -2.3647 -0.1252  1.4096  6.8727
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  37.2851     1.8776  19.858  < 2e-16 ***
## wt           -5.3445     0.5591  -9.559 1.29e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

```
##
## Residual standard error: 3.046 on 30 degrees of freedom
## Multiple R-squared:  0.7528, Adjusted R-squared:  0.7446
## F-statistic: 91.38 on 1 and 30 DF,  p-value: 1.294e-10
```

5.3.2 Graphs

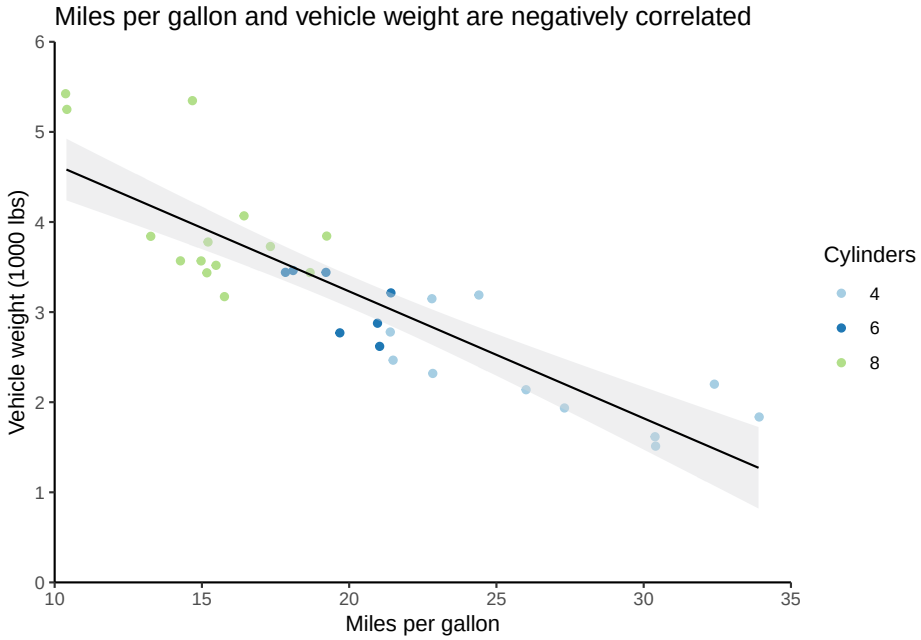
I could've written an entire book on `ggplot`; in fact, someone already has. I'm focusing on the R Markdown piece here, but I've included a lot of information on `ggplot` itself in Section 9.3.

Let's start with a basic scatterplot of the miles per gallon and weight data from the `mtcars` dataset. There are few ways to get the graph from your code chunk into your document. I can just make the graph without saving it as a variable, so it automatically outputs from the chunk, or save it and put the variable name on a new line. This is what I tend to do, as I usually create my graphs in an R script before importing them into my R Markdown document.

```
# Save number of cylinders (cyl) as factor
  # Otherwise, ggplot will treat it as a continuous variable
mtcars <- mtcars %>%
  mutate(cyl = as.factor(cyl))

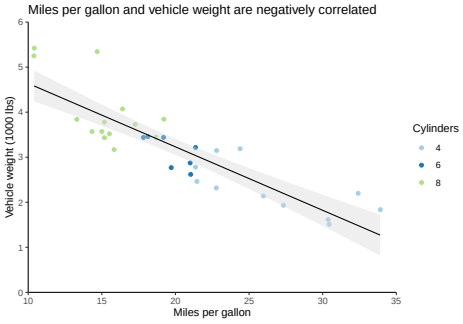
# Create scatter plot
mtcars_scatter <- ggplot(mtcars) +
  geom_jitter(aes(mpg, wt, color = cyl)) +
  geom_smooth(aes(mpg, wt), method = "lm", se = TRUE, level = 0.95,
              fill = "#d7d8db", color = "black", size = 0.5) +
  scale_y_continuous(expand = c(0,0), limits = c(0,6)) +
  scale_x_continuous(expand = c(0,0), limits = c(10,35)) +
  scale_color_brewer(type = "qual", palette = "Paired") +
  theme_classic() +
  labs(title = "Miles per gallon and vehicle weight are negatively correlated",
       y = "Vehicle weight (1000 lbs)",
       x = "Miles per gallon",
       color = "Cylinders")

# Output plot
mtcars_scatter
```



You can use the chunk options to control how this graph appears in the document. The ones I typically use for outputting graphs are dots per inch (`dpi`) to control the image quality, `out.width` and `out.height` with either specific units or a percentage, and `fig.align` to change the alignment of the output. In the graph below, I've set `dpi` equal to 300, the `out.width` at 50%, and `fig.align` to center. You can also add a figure caption in the chunk header if you'd like. These all go in the curly brackets at the top of the chunk and are separated by commas.

```
# Output plot with new chunk options
mtcars_scatter
```



As a note, the values for `out.width` and `fig.align` need to be in quotation marks, while the value for the `dpi` setting doesn't. Pay attention to which values have quotes around them in the column with the default values in the

chunk options section of the reference guide. You'll get an error when you try to render the document if you don't have appropriate quotation marks.

5.3.3 Tables

There are many ways to create tables in R. My preferred way is with `knitr::kable` and `kableExtra`. Together these allow you to create "complex tables and manipulate table styles" as the documentation says. You use the pipe operator `%>%` from `magrittr` the way you use the plus sign for `ggplot` to add layers of formatting. If you don't specify the output format, `knitr` will automatically create an HTML table for you unless you're rendering to a PDF when it will use LaTeX. You can override this locally in `kable` or globally with `knitr::options`.

```
# Get coefficients table from mpg_summary
mpg_coefs <- mpg_summary$coefficients

# Create minimal table
  # Pass table to kable, then format with kable_styling
mpg_coefs %>%
  kable() %>%
  kable_styling()
```

	Estimate	Std. Error	t value	Pr(>|t|)
(Intercept)	37.285126	1.877627	19.857575	0
wt	-5.344472	0.559101	-9.559044	0

This is obviously a very minimal table. The documentation for HTML and LaTeX tables is great, so if you're looking for something in particular it will be relatively easy for you to find. I have a more complex example in Section 6.4.2, to which I added custom CSS code, among other things. You can see what the output of that code would look like in my Psychonomics poster.

5.3.4 Images

It's very easy to embed images. You've already seen this in Section 2.3. Just use the `knitr::include_graphics` function, and you can control the output as you would for a graph by specifying the chunk options. You can also use CSS or `![caption](file/path)` notation, but I find the chunk approach to be more straight-forward.

When I have very complex graphs that take a long time to render, or graphs that don't play very nicely with R Markdown (like those from `corrplot`), I'll save them as images and include them this way.

I also save graphs as images first when I need to crop them to maintain the correct aspect ratio. This is true of the participant maps I created for my

Psychonomics and CNS posters. Instead of using `knitr::include_graphics`, I use `image_read` from the `magick` package to load the image, followed by the `image_trim` function (you can't go straight from a `ggplot` graph to these functions).

5.4 Inline R

A very useful aspect of R Markdown is that you can call R objects and functions in markdown or the YAML header by sandwiching them between backticks. For example, let's say I want to report on the names of the flower species in the `iris` dataset.

```
# Pull species column from iris and get unique values in column
species <- iris %>% pull(Species) %>% unique()

# Print species variable
print(species)
```

```
## [1] setosa     versicolor virginica
## Levels: setosa versicolor virginica
```

Maybe I also want to save a variable with the number of species in this list.

```
# Get number of unique species
species_count <- length(species)

# Print number of species
print(species_count)
```

```
## [1] 3
```

I can call `species` and `species_count` in the markdown text to reference these variables dynamically. Just as in a code chunk, I have to specify that I'm working with R code by including a lower-case r in the backticks with the variable.

So if I put "r species_count" between backticks, I can report that there are 3 species without typing out the number itself. Or I can reference the variable with the exact species names: setosa, versicolor, virginica. You can see what the markdown looks like in the screenshot below.

```
47 So if I put "r species_count" between backticks, I can report that there are `r species_count` species
   without typing out the number itself. Or I can reference the variable with the exact species names: `r
   species`. You can see what the markdown looks like in the screenshot below.
```

You'll also note that `knitr` automatically put commas between the species names. To be honest, this is an aspect of R Markdown that I don't explore very much. I tend to format the variables exactly how I want them to appear in the text before referencing them with inline code. If I find a good explanation of this behavior, I'll link to it here.

Chapter 6

Formatting

There are many ways to customize the formatting of an R Markdown document. In the YAML header, you can specify different parameter options or reference external documents. You can also edit templates directly in certain cases. For local formatting, you can include inline code in the markdown sections.

6.1 YAML parameters

There are several general YAML options that you can include in the YAML header to format your documents. You can also add **params** to the YAML header that you can specify when you render your document and call in your code chunks to make parameterized reports. Some YAML options require quotation marks, while others don't. In general, if the YAML option is a string of text that you're specifying, like the title or your name, then it should be in quotes. If you're setting a programmatic option, like the output type, then it shouldn't be in quotes.

For the string options, there's some custom formatting you can do. The first example below shows how you can center a title and force a line break. The second shows how you can automatically pull the date and time when you knit your document and format it in a particular way. This is actually R code that is embedded in a string by putting the function call between backticks with a lowercase r. You can use the same principle to put R code and functions in the text of an R Markdown document, as described in Section 5.4.

```
---
title: |
  <center> We \"might could\" revisit syntactic processing: </center>
  <center> Studying dialectal variation with event-related potentials </center>
```

```
---
title: "Double modal project analyses: Offline tasks in both dialect groups"
author: "Holly Zaharchuk"
date: "`r format(Sys.time(), '%B %d, %Y')`"
```

```
# Show date formatting code
  # You can run ?Sys.time in the console for more information
  # on the options you can pass to format
format(Sys.time(), '%B %d, %Y')
```

```
## [1] "May 04, 2020"
```

There are also template-specific parameters, but you'll need to look at the specific package documentation to know what these are.

6.2 YAML references

In your YAML header, you can reference other documents for formatting and content.

6.2.1 .bib

To cite references, you need to set the *bibliography* option in the YAML header. I use BibDesk for my reference manager, which creates a .bib file, or you can create a .bib file directly in LaTeX. You can also construct a .bib file through R. See Section 9.1.5 or the .Rmd file for my CV for examples with the `scholar` package.

By default, if you reference a .bib file, the references will appear at the very end of the document. This is usually fine, but sometimes you may want to control the placement of the references, as with a CV. In this case, you can use a .lua filter as described in Section 6.2.4 to place your references in a particular spot.

To create in-text citations, you'll use the cite key from your .bib file with the @ symbol. Full information on citing syntax can be found here. When you cite something from your .bib file, it will appear in your references section when you knit your document. If you want to include all of the references from your .bib file in your reference section, regardless of whether you've cited them or not in the document, set the **nocite** option in the YAML header to **"@*"**.

6.2.2 .csl

To determine the type of formatting for your references, you can include a citation style language or .csl file. There are other ways to set the format of bibliographies, but a .csl file allows fine-grained control over citations that you can also customize. My CV repository has a customized APA 6 .csl file for arranging references in descending order by date. My Psychonomics poster repository also has a customized APA 6 .csl to fix some small bugs.

6.2.3 .cls and .css

You can include .cls files (not to be confused with the .csl files above) for LaTeX styling or .css files for HTML styling. You can see an example of the .cls file that formats my CV in Section 6.3.1. This guide itself uses .css files that you can find here to control the formatting.

6.2.4 .lua

Sometimes, you need to interact with pandoc directly in order to achieve a particular formatting outcome. To do this, you need to use a .lua filter.

The multiple-bibliographies.lua file is incredibly useful. It allows you to use multiple bibliographies in one document, as I do in my CV (one for publications, another for conference presentations). Even if you don't have multiple bibliographies, using this .lua filter will allow you to place your reference section in a particular part of your document.

The .Rmd file for my CV shows how to use multiple bibliographies in R Markdown. First, you need to add a name to each bibliography in the YAML header with an underscore. You can also see how I've referenced this .lua file in the **pandoc_args** option in the YAML header. For this document to knit properly, the .lua file needs to be in the same place as my .Rmd file.

```
name: Holly A. Zaharchuk
position: "Graduate student"
address: "University Park, Pennsylvania, United States"
email: "hzaharchuk@psu.edu"
github: hollzzar
www: sites.psu.edu/bildlab
date: "`r format(Sys.time(), '%B %Y')`"
output:
  vitae::awesomecv:
    pandoc_args: --lua-filter=multiple-bibliographies.lua
bibliography_main: my_cv.bib
bibliography_present: my_presentations.bib
nocite: "@*"
docname: "Curriculum vitae"
csl: apa6.csl
```

Then, you can place the reference section for each bibliography in the appropriate place. This example also shows how you can alter the formatting locally for just the reference section with LaTeX commands.

```
# Presentations

\setlength{\parindent}{-0.2in}
\setlength{\leftskip}{0.2in}
\noindent

::: {#refs_present}
:::

```

Another very useful .lua file handles in-text APA citations. By default, pandoc uses "&" for in-text citations where APA would require "and," even if you're using a .csl file for APA formatting. Rather than downloading a file like *multiple-bibliographies.lua*, this file is already stored with pandoc: you just need to reference it. Follow these instructions to access and reference this .lua filter The example below from my Master's thesis shows how you can reference multiple .lua filters in one document.

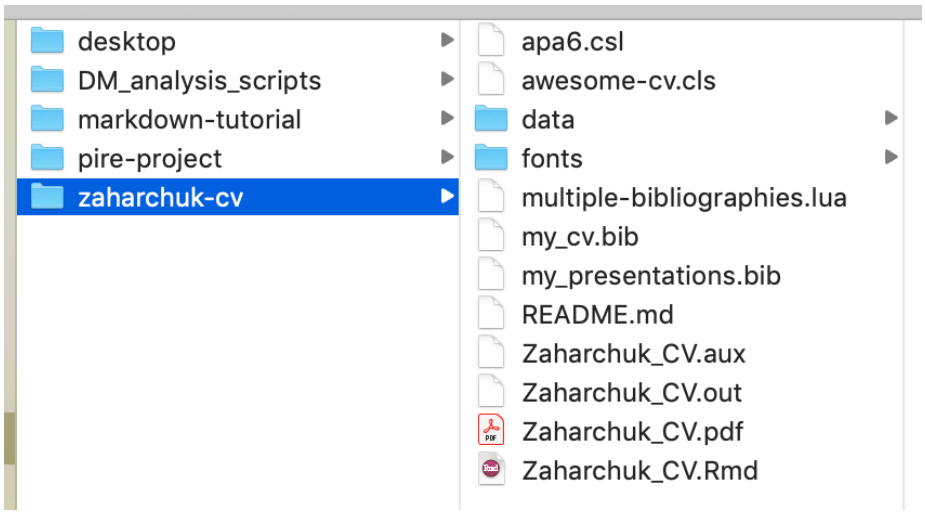
```
---
citeproc: no
output:
  pdf_document:
    fig_caption: yes
    number_sections: yes
    keep_tex: true
    pandoc_args: [
      "--lua-filter", "multiple-bibliographies.lua",
      "--lua-filter", "/Library/Frameworks/R.framework/Versions/3.5/Resources/library/rmdfiltr/replace_amper
sands.lua"
      ]
```

6.3 Editing templates

To make extremely custom edits to templates, sometimes you have to edit the template documents directly. If the template generates a style document (e.g., .cls) in the directory with your .Rmd file, you can usually edit that without going to the package. Otherwise, you have to find out where your computer stores your R packages and edit the template there.

6.3.1 Directory documents

Some templates output formatting documents in your working directory. These are easier to access and edit. For example, knitting the `vitae::awesomecv` template created a .cls file that I could edit to change font sizes/colors.



6.3.2 Package documents

To find out where your packages "live," you can call `installed.packages`. When you leave the parentheses blank (i.e., if you don't provide any arguments), as I've done below, the function will return all of the packages you have installed. I've stored them as a dataframe, and then filtered for an example. Here, I'm looking at the `posterdown` package.

```
# Make dataframe with installed packages
pkgs <- installed.packages() %>%
  as.data.frame()

# Pull posterdown package
pstr <- pkgs %>%
  select(Package, LibPath, Version, Depends, Imports) %>%
  dplyr::filter(Package == "posterdown")

# Make table
kable(pstr) %>%
  kable_styling(bootstrap_options = "condensed", full_width = FALSE, font_size = 12)
```

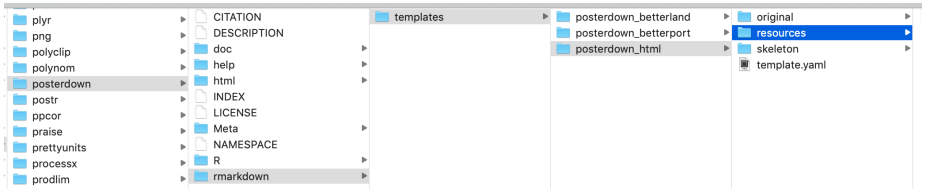
Package	LibPath	Version	Dep
posterdown	/Library/Frameworks/R.framework/Versions/3.5/Resources/library	1.0	NA

To edit package documents once you've located them, you should proceed with caution. Be sure to:

- Save the original template and move it to a different location (in the example below, I made the "original" folder and put the original template

there)

- Make one change at a time, and then re-knit the document to see what changed
- Name the updated template with the same name in the same place as the original (here, in the "resources" folder)



6.4 Inline code

R Markdown supports inline code for custom formatting.

6.4.1 LaTeX

In PDFs, you can use code, typesetting commands (e.g., \vspace{12pt}), and specific packages from LaTeX. There are useful lists of symbols here and here. Check out Writing Your Thesis with R Markdown and Section 6.4.1 below for examples using LaTeX packages and typesetting commands.

I've included examples of calling packages in the YAML header and using inline functions from my statistics homework below.

```
2  output: pdf_document
3  header-includes:
4  - \usepackage{fancyhdr}
5  - \usepackage{color}
6  - \usepackage{makecell}
7  - \usepackage{amsmath}
8  - \usepackage{float}
9  - \usepackage{siunitx}
10 - \usepackage{booktabs}
```

```
27 \pagestyle{fanc
28 \fancyhead[LE,R
29 \fancyhead[LO,R
30
31 ##Problem Set 1
```

PSY 508 Holly Zaharchuk

Problem Set 12

a. *Conduct a logistic regression using all four predictors of suicide. Interpret the coefficients and provide a brief APA-style summary including confidence intervals around the odds ratios (no need to check assumptions here). As always, include a 5 o'clock news summary.*

Model comparison statements

Model A: $ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = \beta_0 + \beta_1(\text{age}_i) + \beta_2(\text{sex}_i) + \beta_3(\text{depression}_i) + \beta_4(\text{SES}_i) + \epsilon_i$
Model C: $ln\left(\frac{\hat{p}}{1-\hat{p}}\right) = \beta_0 + \epsilon_i$
H_0: $\beta_1 = \beta_2 + \beta_3 = \beta_4 = 0$
degrees of freedom: $df = n - pa = 197 - 5 = 192$
additional parameters: $pa - pc = 5 - 1 = 4$

6.4.2 CSS/HTML

In HTML documents, you can include CSS commands. I have some examples in these `revealjs` slides, where I wanted to left-align slide text while keeping titles centered.

Here are some examples of using CSS from my Psychonomics poster. The first example includes custom CSS in a `knitr::kable` table to add borders of a particular color between some rows. The second example changes the font size of the references section at the end of the poster.

```
114  #Knit table
115  knitr::kable(stim_table) %>%
116    kableExtra::kable_styling(font_size=35) %>%
117    collapse_rows(columns = 1:6, valign = "middle") %>%
118    row_spec(4, extra_css="border-top: 5px solid #d7d8db; border-bottom: 5px solid #d7d8db") %>%
119    row_spec(3, extra_css="border-top: 5px solid #d7d8db") %>%
120    footnote(general="ERP time-locked to second modal (could or should) in double modal sentences
     to compare to standard single modal",
121             general_title="")
```

```
428 # References and acknowledgements
429
430 <font size=3><div id="refs"></div></font>
```

6.5 Custom functions

If you're fairly comfortable with R, you can write formatting functions for yourself to make your life easier when you're referencing variables in your text.

When I was working on my Master's thesis, I wrote some custom functions for statistical values that I had to report over and over. This was particularly useful when I had to switch from referring to any p values over .05 as $p > .05$ to p = the exact value. I just had to change the one function by commenting out a couple lines and re-knit my document. The function is included below; this was

the most complicated of the ones I wrote, which included degrees of freedom, t and F values, reaction times, and percentages.

```
# Make number formatting function for p values
# val is the p value I want to format
# format_code specifies whether I want to include the symbol or not
  # format_code defaults to including the symbol
p_formatting <- function(val, format_code = 1) {

  # If I want to include the symbol (when I'm calling this variable in text)
  if (format_code == 1) {

    # If the p value is less than .001, get the less than symbol
      # Otherwise, get the equals symbol
      # The next line, which I commented out for my thesis,
      # looks to see if the value was greater than .05
    sign_type <- if_else(val < 0.001, "<", "=")
    sign_type <- if_else(val > 0.05, ">", sign_type)

    # If the p value is less than .001, set the value equal to .001
    if (sign_type == "<") {
      val <- ".001"

    # If the p value is greater than .05, set the value equal to .05
      # I commented out these next two lines for my thesis
    } else if (sign_type == ">") {
      val <- ".05"

    # Otherwise, get the actual p value,
      # round it to three decimal places, and
      # remove the leading zero
    } else {
      val <- sprintf("%.3f", val)
      val <- substring(val, 2)
    }

    # Combine the new/formatted p value with >, <, or =,
      # depending on the p value
    val_string <- paste(sign_type, val, sep = " ")
    val_string

  # If I don't want the symbol (as in a table),
    # just round the value to three decimal places
    # and remove the leading zero
  } else if (format_code == 0) {
```

```
    val <- sprintf("%.3f", val)
    val_string <- substring(val, 2)
    val_string

  }

}
```

I've included an example with this function below to demonstrate how you can really streamline your analysis-to-presentation pipeline with a simple function.

```
# Create arbitrary p value
p_value <- 0.0123

# Format p value including symbol
p_value_1 <- p_formatting(p_value)

# Format p value not including symbol
p_value_0 <- p_formatting(p_value, format_code = 0)
```

Here is the original p value: 0.0123. I can write $p = .012$ to reference the variable that already has the symbol, or I can reference the variable without the symbol: .012. I've included an image of these lines below to give you a sense of how this looks in markdown.

```
120 Here is the original *p* value: `r p_value`. I can write *p* `r p_value_1` to reference the variable that
    already has the symbol, or I can reference the variable without the symbol: `r p_value_0`. I've included an
    image of these lines below to give you a sense of how this looks in markdown.
```

Chapter 7

Organization

If you're working on a large R Markdown project, like a thesis, it's inevitable that you will have several code chunks to perform pieces of one process (e.g., reformatting data for plotting and then creating the plots), and you will have lots of chapters/sections of markdown.

At a minimum, you want to keep your chunks small and name them for easy troubleshooting and use headers at multiple levels. One thing to note is that you cannot have the same chunk names anywhere in your document or sub-documents; this will throw an error. If you go to the bottom left-hand corner of your .Rmd file in RStudio, you'll see a small drop-down menu. There, you can jump to specific headers and code chunks.

As your documents grow, however, even these sections will become difficult to keep straight. This is when you want to use what are called *child* documents. This just means that you have separate R Markdown documents that you reference in your main R Markdown document. These are also great if you need to include the same code or text in multiple documents.

7.1 `child` documents

The screenshot below shows some code from my main Master's thesis R Markdown document. In this example, I referenced another R Markdown document called *Load*, as well as one called *chapter1*. All you have to do is make a code chunk in which you set the `child` parameter equal to the file that you want to reference (and file path if it's in a subfolder, as I have here). The chunk itself must be empty.

```
<!-- Load all data for both groups -->

```{r child = 'sections/code/Load.Rmd'}
```

\newpage
\pagenumbering{arabic}

<!-- Introduction -->

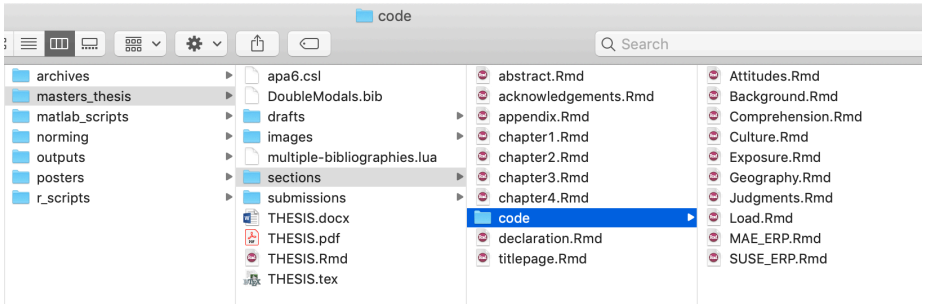
\newpage
\fancyhead[CO,CE]{Introduction}

```{r child = 'sections/chapter1.Rmd'}
```
```

The *Load* document contained several code chunks for loading and processing my data and analyses for presentation. I've included a screenshot of the top of this document below. You can see that I just gave it a **pagetitle** in the YAML header, and then included the code chunks that I wanted. This helped keep the main document uncluttered.

One thing to note is that when you're running your code chunks to work on your document, if you just run the code chunk referencing the `child` document, nothing will happen. You have to open that R Markdown document and run it to evaluate those chunks.

```
---
pagetitle: "Load"
---

```{r offline_vars}
# Set global variables
code_geo <- FALSE
new_image <- FALSE
new_erp <- FALSE
SUSE_tab <- FALSE
new_corr <- FALSE
data_path <- "~/Mirror/DM_analysis_scripts/r_scripts"
writing_path <- "~/Mirror/DM_analysis_scripts/masters_thesis/sections/code"
```

You also saw that I loaded the *chapter1* `child` document. This allowed me to separate out each of the sections of my thesis into different documents, in addition to separating out the analyses as with the *Load* document. I organized these documents into subfolders to keep everything clean. I've included a screenshot of the organizational structure below.

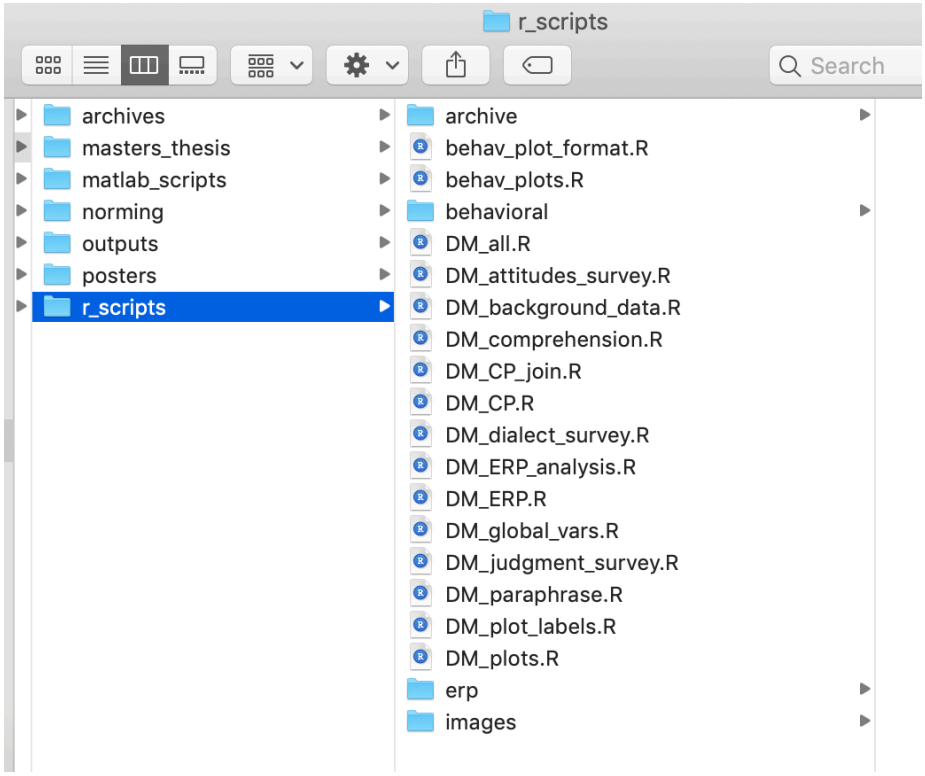
At the top level, I had the main R Markdown document and necessary reference documents (see Section 6.2 for more information on including reference documents in the YAML header). In the first subfolder, I had the documents for each section of the thesis (e.g., *chapter1* is the introduction). In the lower subfolder, I had documents that only contained code chunks (e.g., running statistical analyses and formatting the values for referencing).
```



7.2 Sourcing code

I also use a similar decentralized organizational structure for my actual R processing scripts. You can use `source` to import all of the code from one script into another, similarly to using `child` documents in R Markdown. The *Load* document above actually uses code chunks to load data from external R scripts into the thesis document.

I've included a screenshot of the folder with all of my R scripts for preparing my thesis data. The data themselves are in the *behavioral* and *erp* subfolders. The *DM_global_vars* file contains variables, like the custom formatting functions I described in Section 6.5, that I used across the scripts. I recommend having such a file that you `child` at the lowest level of the importing hierarchy. That way, if you need to change something, you can make that change in just one place.


r_scripts
Search
archives
masters_thesis
matlab_scripts
norming
outputs
posters
r_scripts
archive
behav_plot_format.R
behav_plots.R
behavioral
DM_all.R
DM_attitudes_survey.R
DM_background_data.R
DM_comprehension.R
DM_CP_join.R
DM_CP.R
DM_dialect_survey.R
DM_ERP_analysis.R
DM_ERP.R
DM_global_vars.R
DM_judgment_survey.R
DM_paraphrase.R
DM_plot_labels.R
DM_plots.R
erp
images

Chapter 8

Troubleshooting

1. Warnings vs. errors
2. RStudio vs. R Markdown environments
3. Package issues
4. Strategies for troubleshooting issues

8.1 Warnings vs. errors

8.1.1 Warnings

Warnings won't stop your document from compiling, but generally indicate that you should change something in your code.

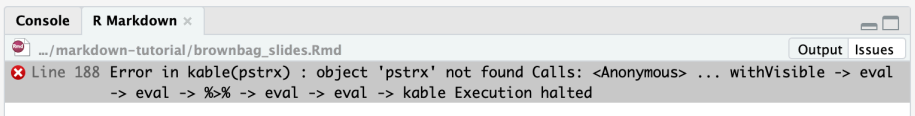
```
> head(dplyr::data_frame(iris),5)
# A tibble: 5 x 1
  iris$Sepal.Length $Sepal.Width $Petal.Length $Petal.Width $Species
              <dbl>        <dbl>         <dbl>        <dbl> <fct>
1               5.1          3.5           1.4          0.2 setosa
2               4.9          3             1.4          0.2 setosa
3               4.7          3.2           1.3          0.2 setosa
4               4.6          3.1           1.5          0.2 setosa
5               5            3.6           1.4          0.2 setosa
Warning message:
`data_frame()` is deprecated, use `tibble()`.
This warning is displayed once per session.
```

8.1.2 Errors

There are different kinds of errors that you can encounter. In general, you can distinguish chunk errors from R Markdown errors by their appearance. Errors will usually tell you which line of the document they're in, but the line numbers

can be misleading or inaccurate, so try to understand the error message first before going to look for the source. As always, Google and Stack Overflow are your friends!

Chunk error:



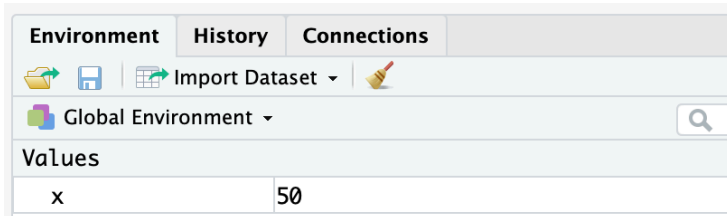
R Markdown error:

```
File troubleshooting.jpg not found in resource path
Error: pandoc document conversion failed with error 99
Execution halted
```

8.2 Environments

Running a chunk executes the code in the console and adds the output to your R environment; however, your R environment is separate from the environment created when knitting a document. When you render your document, it ignores your R environment. `knitr` runs all of the chunks in order and knits them into the markdown text.

If you're getting an error message that says a particular variable or package doesn't exist when it's loaded in your R environment, it's usually because you haven't included it in a previous chunk.




```
# Define new variable y
y <- 100

# When I run this chunk, I get the expected output (150),
# but it fails when I try to knit the document
# I've set eval=FALSE for this chunk, so it doesn't try to run and prevent my document from knitt
print(x + y)
```

8.3 Package issues

8.3.1 Package specification

If a function you want to use isn't included in `base` R, you need to load the package that it's in. The most basic way to do this is to run `library(package)` for each individual package. As you saw in my "setup" chunk example in Section 2.3, my preferred way is to set a list of packages and then load them with the `p_load` function from the `pacman` package. This way, you can easily load multiple packages at once.

If you don't want to load a whole package, you can use this notation: `package::function`. This tells R which package to look in. You especially want to do this if there are what are called "namespace conflicts," where multiple packages have the same function name. I often run into this issue with the `filter` function from `dplyr`, so I always specify `dplyr::filter` even if I've already loaded `dplyr`.

8.3.2 Updates

You may get warnings or errors related to packages being out of date or certain functions being "deprecated." You may need to update packages and software that interact with R and R Markdown to get your code to run and documents to knit. Depending on the feature that is out of date, there are different strategies for updating.

- Update your TeX distribution from the command line
- Update all packages (including `rmarkdown`) in library with `update.packages(path)`
- Update individual packages by reinstalling with `install.packages("package")`
- Update R in the console with the `updateR` package
- Redownload RStudio to update

8.4 Strategies

If you're running into any issues with rendering your document, here are some things you can try:

- Reset your R environment
  - Clear all variables by running `rm(ls = list())` in the console
  - Restart your R environment with CTRL/(control + fn) + shift + F10
  - Run all chunks individually **in order** before compiling to test code
- Search for information
  - Use Help window
  - Search for package in console with `?package` or `??package`
  - Google the error you're getting with the package or function you're trying to use

Chapter 9

Supplementary materials

9.1 Beginner R

- Basic R functionality
- Reading in data
- Tidy R philosophy
- Manipulating data with core `tidyr` functions

9.1.1 Basic R functionality

- Variables
- Functions
- Operators

9.1.2 Variables

The way R stores your information will determine the kinds of functions/operators you can use

- Data: dataframes
- Values: lists, vectors, matrices, etc.

```
# Variables can be numbers, strings, etc.
some_val <- 200
other_val <- TRUE

# Variables can also be lists of numbers, strings, etc.
a_list <- c("a", "b", "c")

# We can also make dataframes (which are essentially special lists)
```

```
a_dataframe <- tibble(column_1 = 1:3,
                      column_2 = a_list)
```

9.1.3 Functions

Functions take a certain number and certain types of "arguments"

- `base` R functions: part of downloading R
- Packages: need to be installed, then loaded
  - `install.packages(package)` once (or to update package)
  - `library(package)` every R session

Use Help window or `?package` to check argument names, types, and defaults

- Named arguments without values are required (and user-defined)
- Named arguments with values show defaults
- Elipses mean that you can add other optional arguments

```
# file: required
# header = TRUE: default
# ...: other potential arguments, like stringsAsFactors = FALSE
read.csv(file, header = TRUE, sep = ",", quote = "\"",
         dec = ".", fill = TRUE, comment.char = "", ...)
```

9.1.4 `base` R

```
# base R function
# Count the number of rows in this dataframe
nrow(a_dataframe)
```

```
## [1] 3
```

```
# If I give nrow() too many arguments, it will throw an error
nrow(a_dataframe, a_list)
```

```
# If I give it the wrong kind of argument, it will just return NULL
# Some functions won't run at all with wrong kind of argument
nrow(a_list)
```

```
## NULL
```

9.1.5 Packages

```
# scholar package
library(scholar)

# get_publications function
# Pull publications from Google Scholar for Marie Curie
```

```
get_publications("EmD_lTEAAAAJ&EmD_lTEAAAAJ&") %>%
  dplyr::filter(cites > 30) %>%
  distinct(title, .keep_all = TRUE) %>%
  select(author, title) %>%
  head(2) %>%
  kable()
```

author	title
P Curie, M Sklodowska-Curie	Sur une substance nouvelle radio-active, contenue dans la pechblende
E Curie	Madame Curie: a biography

9.1.6 Operators

- Relational: >, <, ==, !=, <=, >=
  - `is.na`, `exists`, etc. will return TRUE/FALSE values
  - `grep`, `filter`, `str_detect`, etc. use TRUE/FALSE values
- Logical: !, &, &&, |, ||
- Arithmetic

9.1.7 Reading in data

General parameters for csv files

```
read.csv("file_name.csv",
         header = TRUE,
         stringsAsFactors = FALSE,
         check.names = FALSE,
         na.strings = "")
```

Avoid special characters (including spaces) in file names, directories, and column headers!

- `readxl` package for Excel spreadsheets
- APIs for direct access to online data
  - `qualtRics` package for Qualtrics data
  - `ggmap` package for Google services (geolocation data)
- `read_table` from `readr` package for text files

9.1.8 Tidy R philosophy

- One variable per column
- One observation per row

9.1.9 Manipulating data with `tidyr`

- `%>%`: pass the results of one function on to another
- `select`: choose columns by name

- `mutate`: add/change columns
- `filter`: filter for (or out) rows
- `group_by` and `summarise`: perform operations on groups of data
- ~~`gather and spread`~~ `pivot_longer` and `pivot_wider`: condense multiple columns into one or the inverse
- `separate` and `unite`: split a column into multiple or the inverse

9.1.10 Other helpful `tidyr` functions

- `slice`: choose a row
- `pull`: choose a column
- Helper functions for `select` (e.g., `contains`)
- `join` family of functions: combine datasets based on a shared unique identifier
- `union`: combine datasets by rows (column names must be the same)
- `replace_na`/`drop_na`: alter/remove rows with NA values

9.1.11 Helpful `base` R functions

- `rbind` and `cbind`: add rows/columns
- `nrow` and `ncol`: count rows/columns
- `unique`: pull unique values
- Indexing with `var$column` and `var[row, column]`
- `which` with column/row indexing

9.1.12 Other packages and functions

- `tibble` package for dataframes with `tibble`
- `kableExtra` for `kable` tables
- `ggplot2` package for graphs (cheat sheet here)
- `factor` for ordering text labels in graphs
- Use `na.rm = TRUE` argument (e.g., in `mean`) to remove NA values from calculations

9.2 Intermediate R

- Dealing with free-response text
- Dynamic variable creation and reference
- Helpful functions
- Other tips

9.2.1 Free-response text

- Regular expressions (cheat sheet here)
- Pattern matching
  - Return position/value of elements that match a pattern: `grep`, `agrep`

  - Return TRUE/FALSE: `grepl`, `agrepl`, `str_detect`
  - Change a pattern: `sub`, `gsub`, `replace`
  - Return position of pattern for all elements: `regexpr`, `gregexpr`, `regexec`
- Use `perl = TRUE` argument to handle especially complex patterns

```
# List of elements
fruit <- c("apple", "banana", "pear", "pinapple")

# grep position
grep(pattern = "le", x = fruit)
```

```
## [1] 1 4
```

```
# grep value
grep(pattern = "le", x = fruit, value = TRUE)
```

```
## [1] "apple"    "pinapple"
```

```
# agrep (can also specify value = TRUE)
agrep(pattern = "le", x = fruit, max.distance = 0.1)
```

```
## [1] 1 3 4
```

```
# regexpr
# match.length attribute gives starting position of match
# index.type attribute gives length of matched text
regexpr(pattern = "le", text = fruit)
```

```
## [1]  4 -1 -1  7
## attr(,"match.length")
## [1]  2 -1 -1  2
## attr(,"index.type")
## [1] "chars"
## attr(,"useBytes")
## [1] TRUE
```

9.2.2 Dynamic variables

```
# Variables
vals <- rep(1:3, 3)
name <- "assign_example"

# Assign values to variable name
assign(name, vals)

# Use the variable as usual
assign_example
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
# You can also get the new variable from the name
get(name)
```

```
## [1] 1 2 3 1 2 3 1 2 3
```

```
# You can add to this variable dynamically as well
assign(name, c(get(name), 4:6))

# New output
assign_example
```

```
##  [1] 1 2 3 1 2 3 1 2 3 4 5 6
```

9.2.3 Helpful functions

- `apply`, `lapply`, `sapply`, `tapply`
- `source` variables from R scripts
- `%notin%` and `%in%` (compared to != and ==)

```
# Example using apply: go across columns of dataset
  # and substitute characters
language <- apply(language, 2,
                  function(x) gsub("\\\\", "", x, fixed = TRUE))

# Source other sripts
source("data_cleaning.R", local = TRUE)

# Helper function
"%notin%" <- Negate("%in%")

# Example from processing pipeline for Qualtrics data
unusable <- c("0","00","107")
dat %>% dplyr::filter(Progress==100 & ID %notin% unusable)
```

9.2.4 Other tips

- Store `ggplot2` parameters in a `list`
- `list` vs. `c`

9.3 Graphing with `ggplot`

Now for some general tips on using `ggplot`. I'll go over the following topics:

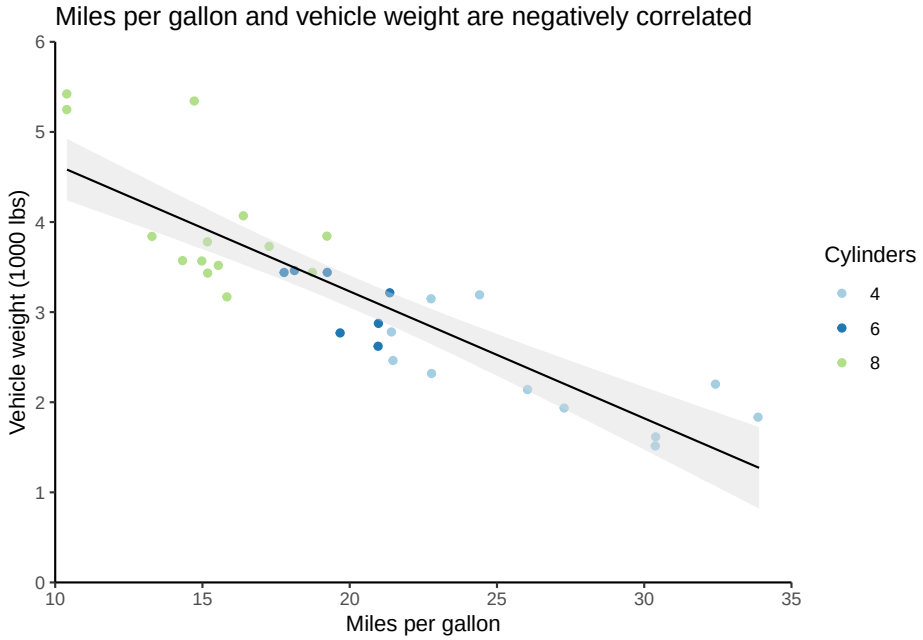
- Formatting
  - Layering plot objects
  - Saving `ggplot` functions in lists

  - Color palettes
  - `color` vs. `fill`
  - Using the `theme` function
- Visualizing
  - Using `Rmisc` for error bars
  - Using `patchwork` to combine multiple plots

9.3.1 Formatting

Recall the scatter plot from Section 5.3.2. If you're playing around with this code yourself, remember that we had to convert `cyl` to a factor so that R would treat it as a discrete rather than a continuous variable.

```
# Output scatter plot
mtcars_scatter
```



A major aspect of `ggplot` that you should be aware of is that the plots are built in sequence. In the graph above (the code also appears below), `ggplot(mtcars)` came first, which just set up a blank graph, then there's `geom_jitter`, which added the `wt` and `mpg` data and point colors based on the `cyl` variable, and so on. If I wanted the regression line from `geom_smooth` to appear beneath the points rather than on top of them, I could've just reordered the function calls. This will also help you troubleshoot if you have issues outputting graphs; just comment out individual plot objects, being careful not to leave any dangling plus signs, and rerun to find the problem.

This functionality also means that you can save several layers in a list to apply

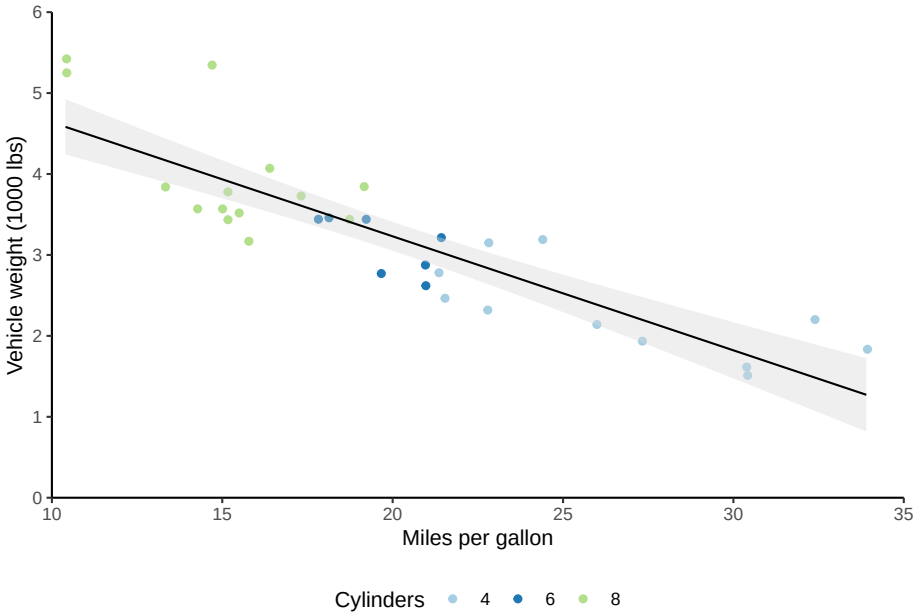
them easily to multiple graphs. This can save you time and effort when you want to have a consistent look across different graphs; just update a particular parameter in your list once, and then it'll apply across your graphs.

Some functions, like `labs`, support this layering very well by allowing multiple calls to the same function. Others, like `scale_y_continuous`, will override previous calls in the same graph (you'll get a warning message when this happens). You'll notice that I have to put commas in between each function in the list rather than plus signs. I also have to save these in a `list`(rather than with `c`).

```
# Create list of specifications
format_list <- list(scale_y_continuous(expand = c(0,0), limits = c(0,6)),
                    scale_x_continuous(expand = c(0,0), limits = c(10,35)),
                    scale_color_brewer(type = "qual", palette = "Paired"),
                    theme_classic(),
                    labs(y = "Vehicle weight (1000 lbs)",
                         x = "Miles per gallon",
                         color = "Cylinders"))

# Create scatter plot
  # Move legend to the bottom
mtcars_scatter <- ggplot(mtcars) +
  geom_jitter(aes(mpg, wt, color = cyl)) +
  geom_smooth(aes(mpg, wt), method = "lm", se = TRUE, level = 0.95,
              fill = "#d7d8db", color = "black", size = 0.5) +
  format_list +
  theme(legend.position = "bottom")

# Output scatter plot with new legend position
mtcars_scatter
```



You'll want to be familiar with the color brewer palettes that I used here. The website I've linked allows you to play with the different options and choose ones with particular characteristics, like color blind-friendly palettes. You can use the `scale_color/fill_brewer` functions with `ggplot` to apply these color palettes.

Relatedly, there's a difference between setting `fill` and `color` options in `ggplot`. Generally, scatter plots get `color` settings, while bar plots get `fill` settings. If I'd set `fill = cyl` in `geom_jitter`, I would've gotten a legend with the three cylinder types, but the colors of the points wouldn't have changed. Similarly, if I'd used `scale_fill_brewer` rather than `scale_color_brewer`, the colors of the points would still have been the R defaults (or you would get an error, depending on the data type).

Lastly on formatting, you'll want to become familiar with the `theme` function as your graphs increase in complexity. The description for this function says it best: "Themes are a powerful way to customize the non-data components of your plots: i.e. titles, labels, fonts, background, gridlines, and legends." I typically layer `theme_classic` with several `theme` specifications. Search `?theme` in your R console or Help window for the full list of parameters.

9.3.2 Visualizing

I often need to make bar graphs to present data. I also tend to want to include error bars on these bar graphs. There are different error bars you can calculate, but I tend to have error bars that represent 95% confident intervals around my means. You'll first need to install the `Rmisc` package to do this, then transform

your data into a long format. I also turned the categorical variables that I cared about into factors. This allows you to control the exact label text and order in your graph. Then, I used the `summarySE` function to create a table with the means and confidence intervals.

```
# Create long-form data for creating error table
  # Mutate columns to create factors with particular labels
  # This is harder to change once you're at ggplot stage
mtcars_long <- mtcars %>%
  select(mpg, wt, cyl) %>%
  pivot_longer(cols = -cyl, names_to = "variables", values_to = "measures") %>%
  mutate(cyl = factor(cyl, levels = c(4, 6, 8),
                      labels = c("4-cylinder", "6-cylinder", "8-cylinder")),
         variables = factor(variables, levels = c("mpg", "wt"),
                            labels = c("Miles per gallon", "Vehicle weight (1000 lbs)")

# Make error table
mtcars_error <- summarySE(mtcars_long, measurevar = "measures", groupvars = c("cyl", "

# Output error table
mtcars_error
```

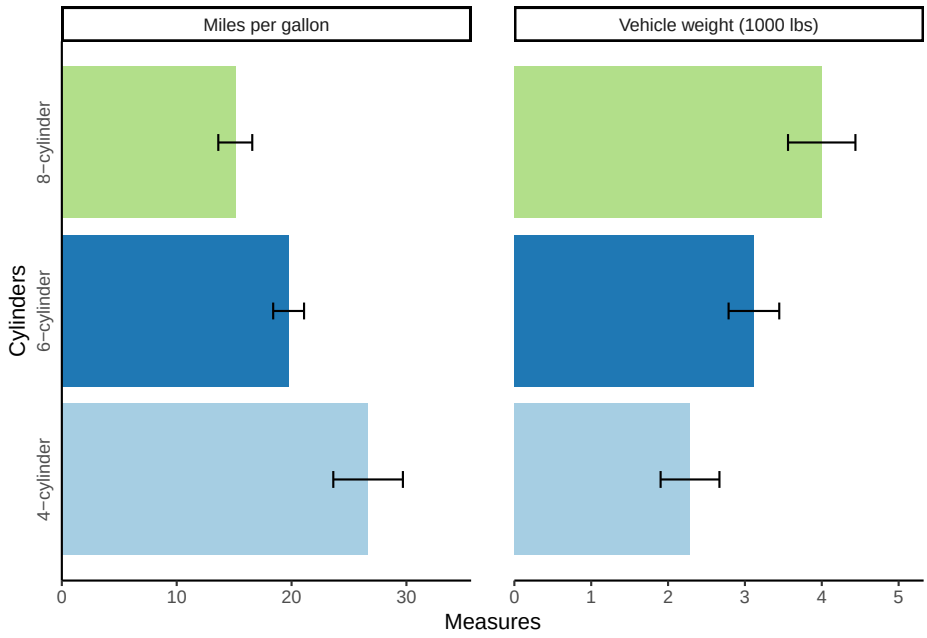
```
##          cyl                 variables  N  measures        sd        se
## 1 4-cylinder          Miles per gallon 11 26.663636 4.5098277 1.3597642
## 2 4-cylinder Vehicle weight (1000 lbs) 11  2.285727 0.5695637 0.1717299
## 3 6-cylinder          Miles per gallon  7 19.742857 1.4535670 0.5493967
## 4 6-cylinder Vehicle weight (1000 lbs)  7  3.117143 0.3563455 0.1346860
## 5 8-cylinder          Miles per gallon 14 15.100000 2.5600481 0.6842016
## 6 8-cylinder Vehicle weight (1000 lbs) 14  3.999214 0.7594047 0.2029595
##          ci
## 1 3.0297434
## 2 0.3826381
## 3 1.3443253
## 4 0.3295647
## 5 1.4781278
## 6 0.4384672
```

Now I can make a bar graph with this error table. My main `ggplot` object uses the `cyl` column and plots the means, while the `geom_errorbar` object uses the confidence interval (`ci`) column to create the error bars. You'll also notice that I changed the width of the error bars and their position, so that they were centered over the bars.

```
# Create bar graph
mtcars_bar <- ggplot(mtcars_error, aes(cyl, measures, fill = cyl)) +
  geom_bar(position="dodge", stat="identity") +
  geom_errorbar(aes(ymin = measures - ci, ymax = measures + ci),
```

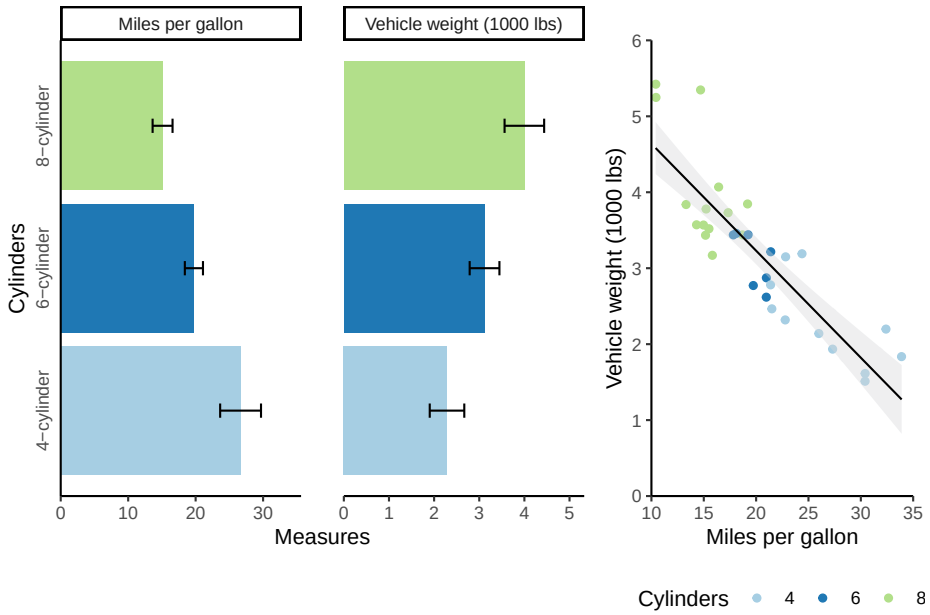
```
                width = 0.1, position = position_dodge(0.9)) +
  coord_flip() +
  scale_y_continuous(expand = expand_scale(mult = c(0, .2))) +
  scale_fill_brewer(type = "qual", palette = "Paired") +
  theme_classic() +
  facet_wrap(~ variables, scales = "free_x") +
  labs(y = "Measures",
       x = "Cylinders",
       fill = "Cylinders") +
  theme(axis.ticks.y = element_blank(),
        legend.position = "none",
        panel.spacing.x = unit(1.5, "lines"),
        axis.text.y = element_text(angle = 90, hjust = 0.5))

# Output bar graph
mtcars_bar
```



Sometimes, you'll want to output multiple graphs with a particular layout. The `patchwork` package allows you to do this easily. The syntax is very similar to `ggplot` syntax, in that you add plots together with a plus sign. You can also add a `plot_layout` object to change aspects of the plots, like their widths.

```
mtcars_bar + mtcars_scatter + plot_layout(widths = c(2, 1))
```



9.4 Practice materials

You can find practice materials for R Markdown and R here. You'll want to clone this repository or download the following documents/folders:

- brownbag_activity.Rmd
- data_cleaning.R
- apa.csl
- /data