Using R Markdown for internal research reports

Holly Zaharchuk

September 17, 2020

Parts of a document

1. YAML header
2. Markdown
3. Code chunks

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 `---` at the beginning and end.

```
---
title: Using R Markdown for internal research reports
author: Holly Zaharchuk
date: "`r format(Sys.time(), '%B %d, %Y')`"
output:
  prettydoc::html_pretty:
    theme: cayman
    highlight: github
---
```

Markdown

The plain text-formatting syntax of R Markdown allows for conversion to multiple document types.

```
# Parts of a document

1. YAML header
1. Markdown
1. Code chunks
```

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.

A full list of chunk options can be found here.

````
```{r setup, include=FALSE, message=FALSE, warning=FALSE}
## R setup ##

# knitr setup
knitr::opts_chunk$set(echo = TRUE, warning = FALSE, message = FALSE)

# Package list
pkg_list <- c("plyr","tidyverse", "ggplot2", "kableExtra", "psych")

# 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 in chunks like you would in an R script by highlighting the relevant lines of code and hitting CTRL/command +

enter
- You can hit the green “play” button in the upper right-hand corner of the chunk
- You can use the **Run** dropdown menu at the top right of your open .Rmd document in RStudio

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.

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

HTML output

For creating research reports, you have a few options:

- `prettydoc::html_pretty`: good-looking documents without trying
- `bookdown::html_document2`: section references with `# Section {#section_name}` and `\@ref(section_name)`
- `html_document`: default

You can easily transform your HTML output to a PDF if you have Google Chrome by running `pagedown::chrome_print(file)` in the console.

You can also copy-paste HTML tables into Word Documents, and they will maintain their formatting.

This is a benefit if you’re collaborating with colleagues who prefer to use Microsoft products, because it is difficult to output even simple tables to Word.

Knit

You can use the **Knit** button or CTRL/command + shift + K to create or “knit” the document.

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.

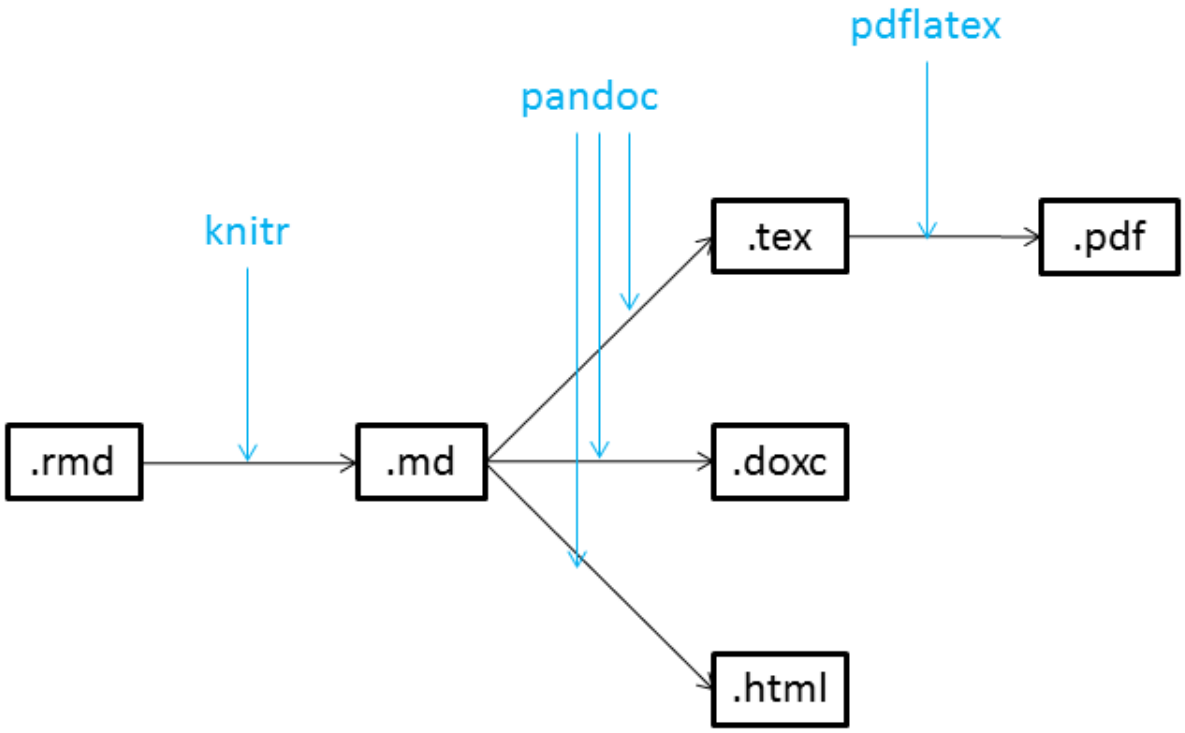
Process

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.



Knitting process from Writing Your Thesis with R Markdown

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.

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**`

- `~~strike-through~~`

There are also `(parentheses)`, `[square brackets]`, and `"quotation marks"` that can have special functions in markdown, like creating hyperlinks: `[text](link)`.

Another note about R Markdown is that line spacing matters.

If you're having issues with your document rendering correctly, make sure you have line breaks between lists, paragraphs, and headers.

Special characters

If you want any special characters for 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, #, \, and $ need to be preceded by a backslash.

You can also engage LaTeX's "math mode" by putting dollar signs around LaTeX math commands.

This way, you can include some LaTeX in R Markdown, even if you're outputting to HTML:

- fractions and binomials
- math symbols
- International Phonetic Alphabet (IPA) symbols

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:

$$e = mc^2$$

Chunk output

Depending on the kind of content you're creating with R Markdown, there are several ways you can take code chunks and turn them into content.

Data

Let's start with the `mtcars` dataset.

If I'm running a regression with `lm` from the `stats` package for example, I can wrap the `summary` function around the output.

This function formats the output cleanly, and also allows me to grab values easily from the coefficients table (this also works with ANOVAs).

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 c
## Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4
## Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4
## Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4
## Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3
## Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3
```

```
# Provide summary statistics for miles per gallon (mpg) and weight (
  # 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
## mpg    1 32 20.09 6.03  19.20   19.70 5.41 10.40 33.90 23.50 0.61
## wt     2 32  3.22 0.98   3.33    3.15 0.77  1.51  5.42  3.91 0.42
```

```
# 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) instea
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
```

Graphs

Let's make a basic scatterplot of the miles per gallon and weight data.

There are few ways to get the graph from your code chunk into your document.

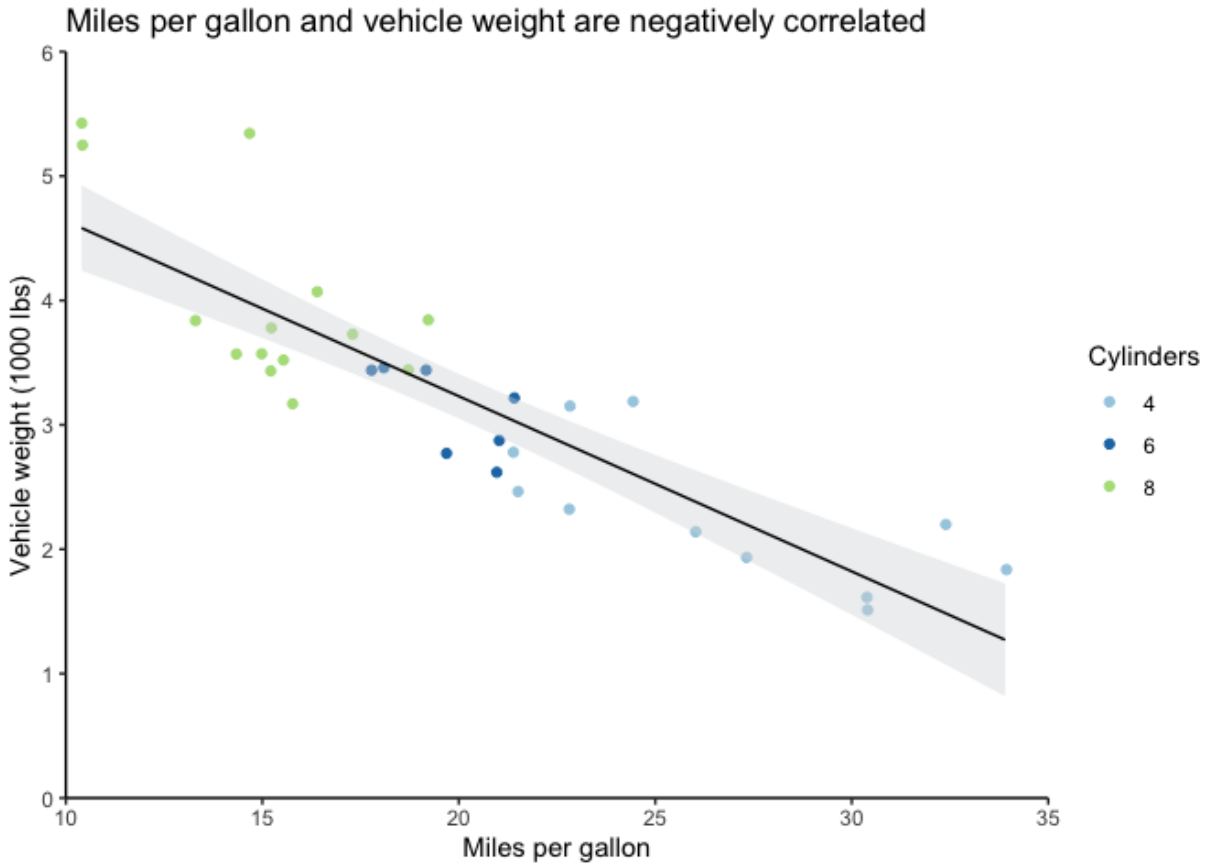
I can just make the graph without saving it as a variable, so it automatically outputs from the chunk, or save it and put the variable name on a new line.

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.

```r
# 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 c
       y = "Vehicle weight (1000 lbs)",
       x = "Miles per gallon",
       color = "Cylinders")

# Output plot
mtcars_scatter
```



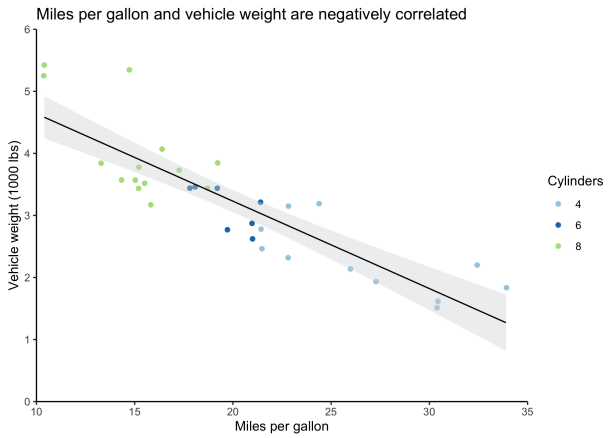
You can use the chunk options to control how this graph appears in the document.

The ones I typically use for outputting graphs are:

- `dpi` to control the image quality
- `out.width` and `out.height`, with either specific units or a percentage
- `fig.align` to change the alignment of the output
- `fig.cap` to add a caption

If you're outputting with `bookdown`, you can also dynamically reference tables and figures with the chunk name— `\@ref(tab:chunk-name)` or `\@ref(fig:chunk-name)` —but be sure that the chunk name doesn't include underscores.

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



As a note, the values for `out.width` and `fig.align` needed to be in quotation marks, while the value for the `dpi` setting didn'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.

Tables

There are many ways to create tables in R, but my preferred way is with `knitr::kable` and `kableExtra`.

These packages work together to allow you to create "complex tables and manipulate table styles" as the documentation says.

You use the pipe operator `%>%` the way you use the plus sign for `ggplot` to add layers of formatting.

```
# 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

You can also use inline CSS to format tables:

```
<style type="text/css">
body{
  font-family: Times New Roman;
  font-size: 10pt;
}
</style>
```

Images

I typically use the `knitr::include_graphics` function to embed images.

You can also use CSS or `![caption](file/path)` notation, but I find the chunk approach to be more straight-forward and customizable.

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.

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).

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)
```

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

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

I can call `species` and `species_count` in the text to reference these variables dynamically.

I have to specify that I'm working with R code by including a lower-case r in the backticks with the variable.

So 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.

```
So I can report that there are `r species_count` species without typ

Or I can reference the variable with the exact species names: `r spe
```

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.

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 can/should be in quotes
- If you're setting a programmatic option, like the output type, then it shouldn't be in quotes

YAML references

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

.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.

By default, if you reference a .bib file, the references will appear at the very end of the document.

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 **"@*"**.

.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.

.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.

.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.

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.

```
---
citeproc: no
output:
  bookdown::word_document2:
    reference_docx: /path/word_template.docx
    pandoc_args: [
      "--lua-filter", "/path/multiple-bibliographies.lua",
      "--lua-filter", "/Library/Frameworks/R.framework/Versions/3.5/
      ]
bibliography_main: bib_main.bib
bibliography_other: bib_other.bib
csl: /path/apa7.csl
---
```

```
# Main references

::: {#refs_main}
:::

# Other references

::: {#refs_other}
:::
```

Organization

At a minimum, you want to keep your chunks small, give chunks unique names for easy troubleshooting/navigation, and use headers at multiple levels.

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.

`child` documents

`child` documents are separate R Markdown files that you reference in your main document.

These are great if you need to include the same code or text in multiple documents.

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.

The chunk itself must be empty.

```
```{r child = 'code/load_data.Rmd'}
```
```

When you run your code chunks while you're working on your main document, you can't just run the code chunk referencing the `child` document like any other chunk.

You have to run that R Markdown file individually, either in the console or by opening it and running the chunks.

Sourcing code

You can use `source` to import all of the code from one script into another, or to import code into an R Markdown document.

```
source("code/scripts/global_variables.R", local=TRUE)
```

I recommend having a file at the bottom of the sourcing hierarchy with basic variables and functions, such as:

- list of packages
- data file names
- HEX codes for graphs/tables
- font size/family for graphs/tables
- custom formatting functions (e.g., *p* values)

That way, if you need to change something, you can make that change in just one place.

Summing up

Tips

- Treat your data as read-only
- Comment code early and often
- Keep code chunks small
- Label chunks uniquely to help with diagnosing issues
- Nest all files under one directory (if possible)
- Don't change the working directory in a chunk

General reference documents

- R Markdown for Psychology Graduate Students
- R Markdown Guide
- R Markdown Cheat Sheet
- R Markdown Reference Guide
- Keyboard shortcuts
- `knitr` documentation