# Communicate with your Audience

## Intro to RMarkdown

Joey Stanley
Doctoral Candidate in Linguistics, University of Georgia
joeystanley.com
orcid.org/0000-0002-9185-0048

Presented at the UGA Willson Center DigiLab
Friday, March 9, 2018

This is the eighth installment of the R workshop series in Spring 2018 and introduces RMarkdown. This document will cover these introductory topics: (1) an introduction to the tool and why it might be handy; (2) the "narrative" or the non-code portion of the document and how to format it; (3) inserting code and how to modify code block; and (4) output formats and ways to modify how the final document looks.

Download this PDF from my website at
*joeystanley.com/r2018*

An Introduction to R: Part 2
by Joseph A. Stanley is licensed under a
Creative Commons Attribution-ShareAlike 4.0 International License.

# 1 Introduction

## 1.1 What is R Markdown?

So far, most of what you've done in R has been with R scripts, which is perfectly fine. But sometimes if you're working on a particular project you might end up with one very long script. You might add some structure to the script using headers, and you can add comments to explain what code does, but after a while it can get unwieldy because there's no *narrative* in your script. You forget why do did what you did and what things you learned from it. For many of my projects, I have an accompanying file where keep all my notes, kind of like a journal of what I did that day. But even then, there's often a disconnect between what's in my notes and what is in my R script.

R Markdown is a good solution to this problem. Essentially, it's a way to combine notes and an R script in one file. The benefit of this is that you can type all the prose you want and then put some relevant block of R code and all the output will be right there. Let me give you several examples of things I do in R Markdown.

1. When I'm starting a new project, I do a lot of exploratory analysis. I create lots of plots and run lots of statistical models trying to get a feel for what I can find in my data. These R scripts tend to be very long because I often create lots and lots of plots and when I go through the code I can never remember which one is the one I want and waste a lot of time executing code just to find the plot I want.

   With R Markdown, I can organize all this into one coherant file. I turn this into a journal by explaing what I'm doing as I do it. I run code and then leave some commentary, including things I learned, mistakes, and what I want to try next. In the end, I have a long document with lots of code, but I can clearly trace my thought process the whole time.

2. When I have a more or less finished idea, I often write a draft of a research paper in the form of an R Markdown file. I only keep the plots and statistical models I might want to include in a real presentation or publication. This is essentially a cleaned up version of the "journal"-style format above, but instead of recording my thought process as I learn something, it's more of an explanation to an unknown audience. When that is done, the transition to paper or presentation is easier because I already have the draft done. It also saves some headache later because I know exactly where to go if I need to change a plot or something.

3. Another major application of R Markdown is to produce professional-looking documents like the one you're reading right now. In fact, all of the handouts I've done for these workshops were written entirely in R Markdown. When you work with R Markdown, you're working in RStudio, but you then "knit" the files up and they produce HTML files, PDFs, or Microsoft Word files. If you have a way to host them, the HTML files can easily be put online for others to see. Saving the file as a PDFs is very similar to using LaTeX, if that's


orcid.org/0000-0002-9185-0048

the way you want to go. In Word, everything comes with preset formatting, but if you've used styles in Word it's easy enough to change those. Regardless of what format you use, they're automatically formatted and the code, plots, and other output are automatically included, so your methods for producing some plot or analysis are immediately transparent.

4. I'm working on a big project right now with a team of other researchers. The dataset that we're sharing on GoogleDrive is growing still, usually every month but sometimes as often as every week it'll update. With R Markdown I can write a document that will read in the data and do visualizations and other analysis. Every time I run the file, it'll automatically read in the latest dataset and create all the plots and run the statistical models on the newest data all at once. This is a pretty handy way to make sure I'm staying current with my data.

5. I recently did some research and I wanted to summarize what my findings for someone. Using R Markdown, I could create a file that loaded my data and essentially walked through the analysis step by step. I could then share the resulting HTML output as a stand-alone file. That way, they could see the code and plots without me having to share the data itself. (We ended up co-authoring, so they got access the data anyway.)

So R Markdown is quite handy, and makes for a great bridge between working in R and putting together a paper. As far as I'm aware, not too many publishing companies accept files in R Markdown, but some people have written entire books using the slightly more advanced bookdown, like this super meta book on bookdown written in bookdown[1]. You can also make slideshows with RMarkdown and incorporate Shiny elements in them as well.

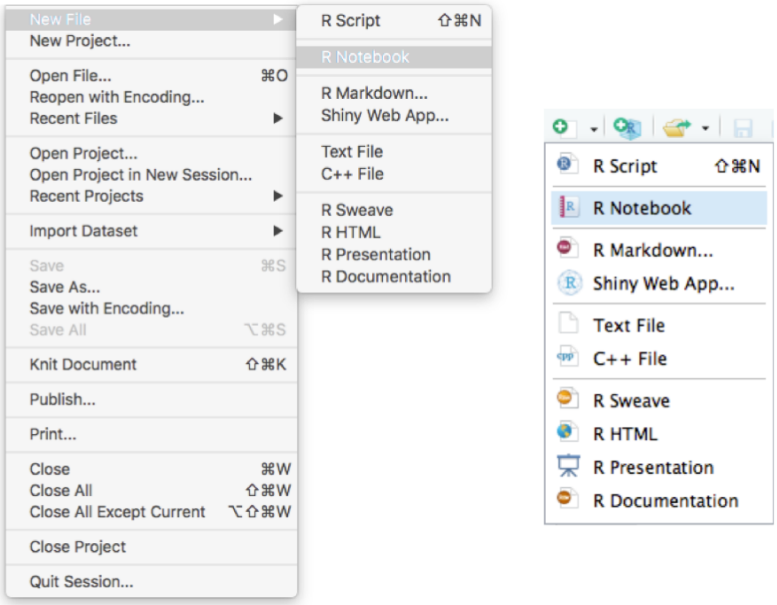
## 1.2 What to expect

Today's workshop will cover the basics to get you going in R Markdown. Most of what makes an R Markdown file so good is whatever R code you want to incorporate into the file, so you're on your own as far as content. We'll divide the workshop into three sections: the narrative, the code, and output. The narrative portion of your document is all the stuff that isn't R code. The code blocks are marked chunks of R code that execute as if they were in an R script. The output is mostly controlled in the header. We'll look at formatting and some of the options that are available for each of these.

## 1.3 Getting started

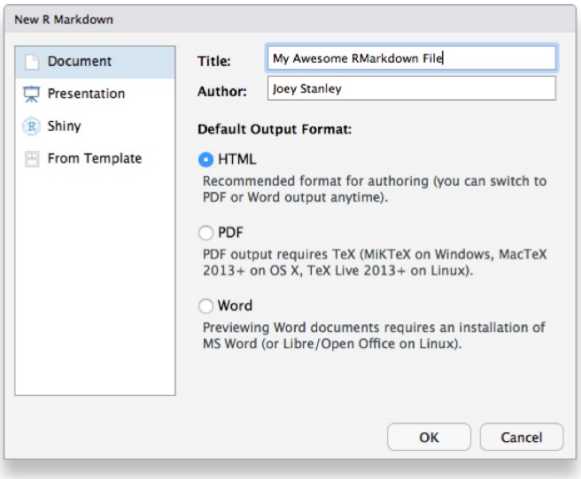
To get started with R Markdown, open RStudio and go to `File -> New File -> R Markdown`. Or, you can click the little downward pointing arrow next to the New Document Icon (the one with the green plus at the top left of RStudio) and choose `R Markdown`.

[1] https://bookdown.org/yihui/bookdown/




If you don't have the required packages for R Markdown, it'll prompt you to install or update. As far as I can tell, the list of essential ones are `rmarkdown`, `formatR`, `caTools`, `bitops`, `tools`, `utils`, `knitr`, `yaml`, `htmltools`, `evaluate`, `base64enc`, `jsonlite`, `rprojroot`, `mime`, `methods`, `stringr`.

Assuming everything works for you, a window will pop up that will ask you for some details about your new file. Give it whatever title you want and you can put your name as the author. For now, let's stick with the default output format, HTML, because it makes things a little bit quicker. On the left pane, we'll just do a Document, but here's how you would start a slideshow or a file with Shiny elements incorporated into it.





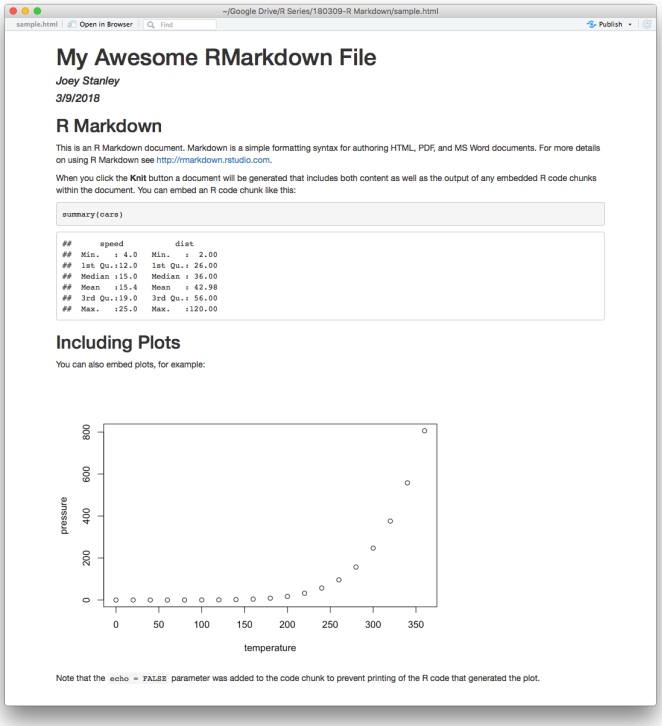
orcid.org/0000-0002-9185-0048

When you click "OK", you'll see a new tab appear in RStudio with some boilerplate code. The top portion will have some metadata in a block of code. If you look closely, this portion has a line with three dashes at the top and bottom of the block.

```
---
title: "My Awesome RMarkdown File"
author: "Joey Stanley"
date: "3/9/2018"
output: html_documente
---
```

This is the *header*. For now, it just contains the metadata we put in at the beginning. We'll get to that at the end of this workshop. The rest of the document contains some example code that explains how the document works and how to get it to run. We'll explain this and much more in this workshop.

Just to give you an idea of what this document looks like, let's "knit" it up as an HTML file. When you have an RMarkdown file open, there should be some new buttons and things to click just above your script (but below the tabs themselves). One of these is a little blue ball of yarn with the label "Knit." Go ahead and click that. (You'll be prompted to save your file if you haven't already.) You'll see R thinking for a little bit, and then a new window will pop up.




As you can see, the title, author, and date are automatically included at the top, and then there's the body of the document. There are various headers, code blocks, output, and narrative. This is

what a finished R Markdown file looks like. It's simple, but it is complete. So now let's dive in and see if we can understand more about what's going on here.

# 2 The Narrative

Let's get started with what is probably the most straightforward aspect of RMarkdown files: the text. I'm calling this portion the *narrative* because it is where you narrate what is happening in your code block (well, I do at least).

R Markdown is actually just an extension of another language called *Markdown*. Both of these are kinds of *markup languages* that lets you add simple formatting to plain text. It's lightweight so it's easy to learn and easy to read documents written in Markdown. It's pretty universal (Reddit and Wikipedia both use it or forms of it) so once you use learn it you can easily transfer skills from one place to another and it's not dependent on some operating system or specific software. Also, because it's so easy and universal, it's predicted to have some decent longevity, which is a big deal in computer code.

The way you include narrative in an R Markdown file is, quite simply, to just start typing. It'll automatically render correctly in the final output. To start a new paragraph, use two spaces at the end of the line or, more simply, just put a blank line between each paragraph in your code. There are just a couple formatting things you might want to add to make the document a little better.

## 2.1 Formatting

Simple formatting is straightforward. To put something in *italics*, put it in asterisks, `*like this*`. To make something **bold**, use `**two asterisks**`. As you might expect, ***bold and italics*** just uses `***three***`. (Alternatively, you can use `_underscores_` instead of asterisks and it'll work just the same.) You'll notice that RStudio automatically changes the color of the text within the file itself, which is handy. But when you knit the file, the color will go away, but the proper formatting will remain.

```
## Formatting

Simple formatting is straightforward. To put something in *italics*, put it in asterisks, `*like this*`. To make
something **bold**, use `**two asterisks**`. As you might expect,  ***bold and italics*** just uses
`***three***`. (Alternatively, you can use `_underscores_` instead of asterisks and it'll work just the same.)
You'll notice that RStudio automatically changes the color of the text within the file itself, which is handy.
But when you knit the file, the color will go away, but the proper formatting will remain.
```

Some other things you might use are ~~strikethroughs~~ which are done with `~~double tildes~~` and superscripts[1] with `^double carats^`. The one I probably use the most is putting something


orcid.org/0000-0002-9185-0048

in a `fixed-width` font, which I use when I use in-line code snippets (like `x <- c(1:10)`) or when I refer to functions by name (like `mutate` or `summary`).

There are a couple special characters you might want to include. The first, em-dashes (—), which are done with three hyphens in a row: `---`, are for parentheticals that you want to pack more of a punch than parentheses. A similar character is the en-dash (–), which can be created with two hyphens `--`, which is used in the place of em-dashes in the UK. They're also used in number ranges, so "3-5" is "three dash five," but "3–5" is "three through five". Finally, you may want to include ellipses (…), which, as you might expect, is just three periods `...` in a row.

## 2.2 Headers

You can organize your document using headers, up to six levels deep. To do this, type the name of the header you want, and put between 1 and 6 pound symbols at the start of the line to correspond to levels 1 through 6:

```
# Header 1

## Header 2

### Header 3

#### Header 4

##### Header 5

###### Header 6
```

Not only do these give your document structure, but they're also used as entries in the table of contents if you want one.

## 2.3 Larger structures

Things like bold and italics are generally used at the individual word level, but for some parts of your document, you may need to format larger chunks of text.

### *2.3.1 Block quotes*

One example is how to include a quote like this:

"The problem, once solved, is simple."

Do do this, all you need to do is start the line with a single "greater than" symbol (`>`):

```
> "The problem, once solved, is simple."
```

This will automatically format it a little differently to make it stand out.

### 2.3.2 Unordered list

You may also want to add a bulleted list. You can do that by starting each line with an asterisk and sub-items with a plus.

```
* Item 1
* Item 2
```

- Item 1
- Item 2

### 2.3.3 Ordered list

You can add a numbered list as well by adding a `1.` at the start of each line. What's cool about these is that you just need to put the number 1 and it'll number them automatically for you. This makes it easy to reorder the items without having to constantly change the numbers manually.

```
1. Item 1
1. Item 2
```

1. Item 1
2. Item 2

One problem I've had is sometimes I want multi-paragraph text as one of the numbered items. But when you do that, the numbers restart:

```
1. Something cool here.

Additional commentary about it.

1. Something else that's cool here.
```

1. Something cool here.

Additional commentary about it.

1. Something else that's cool here.

No, that doesn't seem to work out well. The solution is to indent the new paragraph using four spaces. That'll do exactly what you want it to do.

```
1. Something cool here.

    Additional commentary about it.

1. Something else that's cool here.
```

1. Something cool here.

    Additional commentary about it.


orcid.org/0000-0002-9185-0048

2. Something else that's cool here.

Note that it's not just that the four spaces happen to line up with the start of the paragraph. Instead, RMarkdown interprets the four spaces as a specific formatting code and makes the appropriate changes to that paragraph but also to the numbering. Later on, when we see how to add actual R code, you can do the same trick and insert as much material as you want and it'll work out just as you want it to. Pretty slick.

### 2.3.4 Blocks of code

Finally, sometimes you might just want to put a whole paragraph of fixed-width code. I do this every time you see example code.

```
To do this, just start each line with four spaces or an indent.
```

RStudio interprets that accordingly (and notices you're not in a numbered list), and puts the whole thing indented and in the right font.

## 2.4 Links

Sometimes you want to include links to websites or includes pictures in your files. This section briefly covers how to do that.

### 2.4.1 Hyperlinks

If you want to include *hyperlinks*, you need to include the text that you want to be highlighted and the link you want to send people do. The syntax is to first put the text in square brackets `[ ]` and immediately after that put the full url in parenthesis `( )`.

So the code for *this link* is `[this link](https://en.wikipedia.org/wiki/Hyperlink)`.

### 2.4.2 Images

Images are similar to hyperlinks, but instead of including some text, you put nothing between the square brackets, and instead of a path to a webpage, you put the path to the image on your computer. You also need to put an exclamation point at the start, before the square brackets.

Here's an image:


And here's the code:

```
![](/Users/joeystanley/Google Drive/R Series/Hexagons/rmarkdown.png)
```

### 2.4.3 Your Turn!

**The challenge**

Try to recreate the following formatted text—which by the way is 100% real—using RMarkdown.

***The Bulwer-Lytton Fiction Contest***

Since 1982 the English Department at San Jose State University has sponsored the *Bulwer-Lytton Fiction Contest*[2], a whimsical literary competition that challenges entrants to compose the opening sentence to the worst of all possible novels. The 2017 winner was submitted by Kat Russo of Loveland, Colorado:

The elven city of Losstii faced towering sea cliffs and abutted rolling hills that in the summer were covered with blankets of flowers and in the winter were covered with blankets, because the elves wanted to keep the flowers warm and didn't know much at all about gardening.

And the runner-up was by Tony Buccella of Allegany, New York:

Although in the rusty tackle-box of his mind he yearned to be a #3 buck-tail spinner, Bob knew deep down he must accept his cruel fate as a bottom bouncer rig, forever destined to scrape the muddy bottom of the river of life.

The rules to the Bulwer-Lytton Fiction Contest are childishly simple:

- Each entry must consist of a single sentence but you may submit as many entries as you wish. (One fellow once submitted over *three thousand* entries.)
- Sentences may be of any length **but we strongly recommend that entries not go beyond 50 or 60 words**. Entries must be "original" (as it were) and previously unpublished.
- Surface mail entries should be submitted on index cards, the sentence on one side and the entrant's name, address, and phone number on the other.
- E-mail entries should be in the body of the message, **not in an attachment** (and it would be really swell if you submitted your entries in Arial 12 font).
- The *official* deadline is April 15 (a date that Americans associate with painful submissions and making up bad stories). The *actual* deadline is June 30.
- Finally, in keeping with the gravitas, high seriousness, and general bignitude of the contest, the grand prize winner will receive… a pittance.

[2] http://www.bulwer-lytton.com/


orcid.org/0000-0002-9185-0048

If you would like to submit an entry to the contest, or you would like to contact *the Grand Panjandrum himself*[3], you have two main options.

1. You may email Scott Rice at *srice@pacbell.net* .
2. You may also send your contest entries, requests, and various and sundry truckling to:

   Bulwer-Lytton Fiction Contest

   Department of English

   San Jose State University

   San Jose, CA 95192-0090

Please include your name, phone number, and addresses—Gastropoda and e-mail. (Note: this data is for our contact information, not for public consumption.)

**The solution**

Here is the code for the text.

```
***The Bulwer-Lytton Fiction Contest***

Since 1982 the English Department at San Jose State University has sponsored
the [Bulwer-Lytton Fiction Contest](http://www.bulwer-lytton.com/), a whimsic
al literary competition that challenges entrants to compose the opening sente
nce to the worst of all possible novels. The 2017 winner was submitted by Kat
Russo of Loveland, Colorado:

> The elven city of Losstii faced towering sea cliffs and abutted rolling hil
ls that in the summer were covered with blankets of flowers and in the winter
were covered with blankets, because the elves wanted to keep the flowers warm
and didn’t know much at all about gardening.

And the runner-up was by Tony Buccella of Allegany, New York:

> Although in the rusty tackle-box of his mind he yearned to be a #3 buck-tai
l spinner, Bob knew deep down he must accept his cruel fate as a bottom bounc
er rig, forever destined to scrape the muddy bottom of the river of life.

The rules to the Bulwer-Lytton Fiction Contest are childishly simple:

* Each entry must consist of a single sentence but you may submit as many ent
ries as you wish. (One fellow once submitted over *three thousand* entries.)
```

[3] http://www.people.com/people/archive/article/0,,20084768,00.html

```
* Sentences may be of any length **but we strongly recommend that entries not
go beyond 50 or 60 words**. Entries must be "original" (as it were) and previ
ously unpublished.

* Surface mail entries should be submitted on index cards, the sentence on on
e side and the entrant's name, address, and phone number on the other.

* E-mail entries should be in the body of the message, **not in an attachment
** (and it would be really swell if you submitted your entries in Arial 12 fo
nt).

* The *official* deadline is April 15 (a date that Americans associate with p
ainful submissions and making up bad stories). The *actual* deadline is June
30.

* Finally, in keeping with the gravitas, high seriousness, and general bignit
ude of the contest, the grand prize winner will receive… a pittance.

If you would like to submit an entry to the contest, or you would like to con
tact [the Grand Panjandrum himself](http://www.people.com/people/archive/arti
cle/0,,20084768,00.html), you have two main options.

1. You may email Scott Rice at srice@pacbell.net .

1. You may also send your contest entries, requests, and various and sundry t
ruckling to:

    Bulwer-Lytton Fiction Contest

    Department of English

    San Jose State University

    San Jose, CA 95192-0090

Please include your name, phone number, and addresses---Gastropoda and e-mail
. (Note: this data is for our contact information, not for public consumption
.)
```

Interestingly, the email was automatically converted into a hyperlink without me having to do anything.


orcid.org/0000-0002-9185-0048

# 3 Code blocks

So the narrative is relatively straightforward. However, the reason you're using RMarkdown is because you want to incorporate R code into your text. Fortunately, this is pretty easy as well. Plus, it comes with all sorts of options so you can customize how it renders in the final product.

To include a code block, all you need to do is start and end the block with a line of three tick marks (`), that thing that shares a key with the tilde next to the number 1 key. After the three on the top one, you'll need to put `{r}` so that RMarkdown knows to interpret what's inside as R code. This means you'll also get syntax highlighting, which is always a plus.

Thus, this code

````
```{r}
2 + 2
```
````

produces this output:

```r
2 + 2
```

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

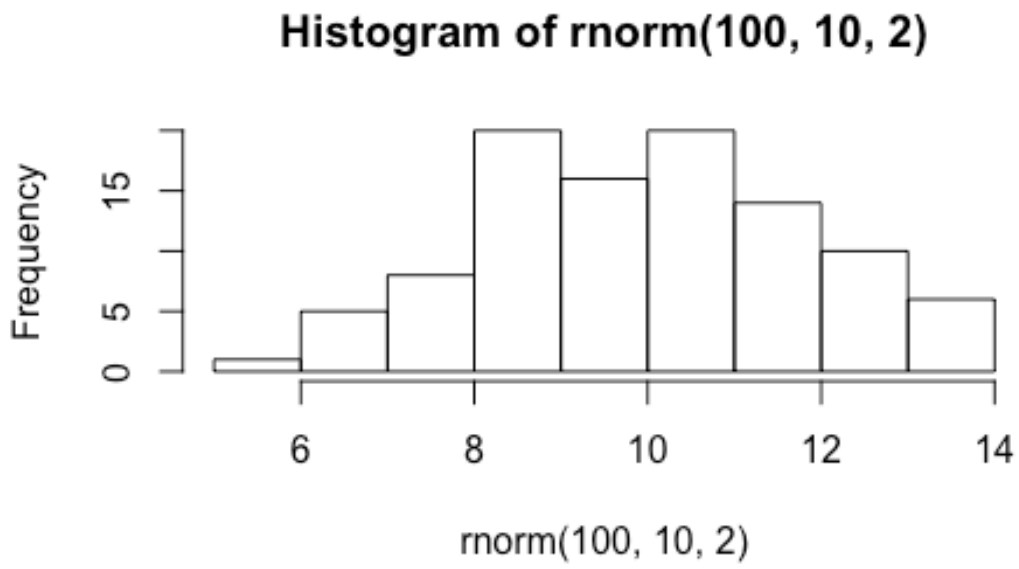
Side note, there's a keyboard shortcut for creating empty code blocks. For a Mac it's option + command + i and for Windows it's control + alt + i. I'd recommend just using that every time.

Also, you can give this code block a name by typing whatever name you want after the `r` in the `{}`. This is handy if you use R Studio's navigation to jump around, but it's not necessary.

The great thing about incorporating code like this is that it is regular R code and you can run it like you would any other R code. If you do and there's some sort of output, like a plot or a summary table, it'll show it to you in the RMarkdown file.

When you include code chunks like these in an RMarkdown file, the code and the output are also displayed in the final product. So if we wanted to include a plot of 100 normally distributed points, we could do so and it would show up.

```r
hist(rnorm(100, 10, 2))
```




Of if you want to include some sort of output, like summary statistics, you can do that as well:

```
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
summary(cereal)
```

```
##                          name    mfr    type       shelf
##  100% Bran                  : 1   G:22   C:74   Min.   :1.000
##  100% Natural Bran          : 1   K:23   H: 1   1st Qu.:1.500
##  All-Bran                   : 1   N: 6          Median :2.000
##  All-Bran with Extra Fiber: 1   P: 9          Mean   :2.227
##  Almond Delight             : 1   Q: 7          3rd Qu.:3.000
##  Apple Cinnamon Cheerios    : 1   R: 8          Max.   :3.000
##  (Other)                   :69
##
```

The code blocks themselves can be modified though so that you have some control over how they look. Let's see what kinds of things you can change.

## 3.1 How the code is run

When you create a code block, there's a space for you to add various options. We've seen this already with the `{r}` at the start of the code block. The part between the brackets is where you put the "chunk options." This `r` tells R to interpret the code as R code and applies the appropriate syntax highlighting. If you wanted to include python code or something, you can actually change that to `python` and it'll do change how the code is highlighted.

The main thing you might need to do to change how the code is run is to make the code *not* run. The way to do this is by adding `eval = FALSE` as a chunk option, after `r`, and separated by a comma:


orcid.org/0000-0002-9185-0048

```
```{r, eval = FALSE}
2 + 2
```
```

This will make the code block appear in your R Markdown file, but it won't run. I use this all the time in these R workshops. For example, I might give an example of bad code that will not run.

```
```{r bad, eval = FALSE}
x = c(1,2,3,4,
```
```

If I didn't include `eval = FALSE`, R will try to run the code when it compiles the document and it'll throw an error. Or I might be illustrating an example and will use generic names:

```
```{r template, eval = FALSE}
variable = function(argument1, argument2)
```
```

If I know something is going to be problematic in that way and I still want to include it, this is a great way to do so.

The last thing you might see is `include = FALSE`. In fact, in R Markdown file that is generated when you create a new file, the first code block has this option. This means that the chunk will run but it won't be included at all in the final document. This is great for setting up global options, but are generally reserved for more advanced settings.

## 3.2 How the results are displayed

### *3.2.1 Hide all results*

The other thing you might want to do with a code chunk is to modify how the *results* get displayed. The main thing is that you might want to hide the results entirely. You can do this by setting the option `results = "hide"`.

So with this block of code, we run a summary of the `cereal` data we read in earlier. If you wanted to, you could run the summary block but not actually display it. So this is the code you would type.

```
```{r, results = "hide"}
summary(cereal)
```
```

And this is what the result would be.

```
summary(cereal)
```

The only application I see for this is if you want to include some output in the R Markdown file itself (like in RStudio as you're typing the code), but not include it in the document. I honestly

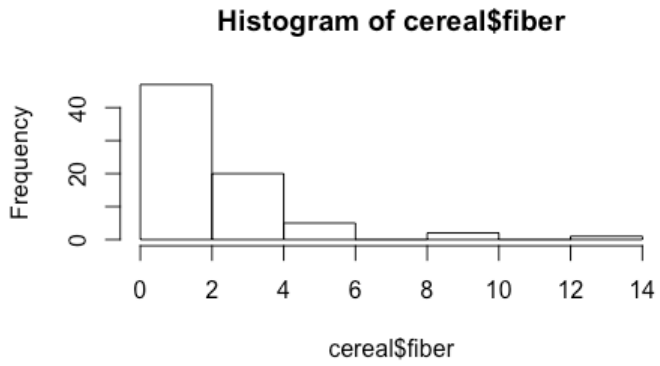
have never used this option and can't think of a really good reason why you might need to. But it's there in case you need it.

### 3.2.2 Hide the code

The opposite of this would be to hide the code but display the output. This is actually super useful for a lot of reasons. The way you do this is to add the option `echo = FALSE` to the options. Here's the code:

```
```{r, echo = FALSE}
plot(cereal$fiber)
```
```

And here's the output:




There are two really good reasons for why you might want to do this. First, sometimes the exact code isn't necessary to include, or sometimes you want to create a document that actually shows something useful from the plots and you don't want the minutiae of the code getting in the way, especially if it's a very long code block.

The other case is when you actually want to display results that are different from what the code shows. I do this all the time when I read in files from my computer, or when I need to print something and I want to show fewer lines than the default. This works well in conjunction with `eval = FALSE` so you have two blocks: one to show the code but without running it, and another to show the results without showing the code.

For example, here's a case where I'm reading in a file from my computer, but the path name is super long and annoying to read and distracting from what I'm trying to do. Also, the output is not what I would want to show because the McDonald's menu item names are super long.

```
read.csv("/Users/joeystanley/Desktop/github/joeystanley/data/menu.csv")
```

```
##     Category                                                  Item
## 1  Breakfast                                       Egg McMuffin
## 2  Breakfast                                  Egg White Delight
## 3  Breakfast                                   Sausage McMuffin
## 4  Breakfast                             Sausage McMuffin with Egg
```


orcid.org/0000-0002-9185-0048

```
## 5  Breakfast                                Sausage McMuffin with Egg Whites
## 6  Breakfast                                          Steak & Egg McMuffin
## 7  Breakfast                  Bacon, Egg & Cheese Biscuit (Regular Biscuit)
## 8  Breakfast                    Bacon, Egg & Cheese Biscuit (Large Biscuit)
## 9  Breakfast Bacon, Egg & Cheese Biscuit with Egg Whites (Regular Biscuit)
## 10 Breakfast   Bacon, Egg & Cheese Biscuit with Egg Whites (Large Biscuit)
##     Oz Calories Fat Sugars
## 1  4.8      300  13      3
## 2  4.8      250   8      3
## 3  3.9      370  23      2
## 4  5.7      450  28      2
## 5  5.7      400  23      2
## 6  6.5      430  23      3
## 7  5.3      460  26      3
## 8  5.8      520  30      4
## 9  5.4      410  20      3
## 10 5.9      470  25      4
```

As you can see that's stupid long and takes up a lot of space, especially if this is going to be on paper. So what I do is I read it in, shorten the menu items to just 24 characters, and print just the first 15. I also shorten the pathname and pretend I keep all my data on the Desktop, which is totally not true.

```
##     Category                      Item  Oz Calories Fat Sugars
## 1  Breakfast              Egg McMuffin 4.8      300  13      3
## 2  Breakfast         Egg White Delight 4.8      250   8      3
## 3  Breakfast          Sausage McMuffin 3.9      370  23      2
## 4  Breakfast Sausage McMuffin with Eg 5.7      450  28      2
## 5  Breakfast Sausage McMuffin with Eg 5.7      400  23      2
## 6  Breakfast     Steak & Egg McMuffin 6.5      430  23      3
## 7  Breakfast Bacon, Egg & Cheese Bisc 5.3      460  26      3
## 8  Breakfast Bacon, Egg & Cheese Bisc 5.8      520  30      4
## 9  Breakfast Bacon, Egg & Cheese Bisc 5.4      410  20      3
## 10 Breakfast Bacon, Egg & Cheese Bisc 5.9      470  25      4
## 11 Breakfast Sausage Biscuit (Regular 4.1      430  27      2
## 12 Breakfast Sausage Biscuit (Large B 4.6      480  31      3
## 13 Breakfast Sausage Biscuit with Egg 5.7      510  33      2
## 14 Breakfast Sausage Biscuit with Egg 6.2      570  37      3
## 15 Breakfast Sausage Biscuit with Egg 5.9      460  27      3
```

So the end result in your code looks like this:

````
```{r, eval = FALSE}
read.csv("/Users/joeystanley/Desktop/menu.csv")
```


```{r, echo = FALSE}
menu<-read.csv("/Users/joeystanley/Desktop/github/joeystanley/data/menu.csv")
menu$Item <- stringr::str_sub(menu$Item, 1, 24)
head(menu, n = 15)
```
````

Of course, this means that it's easy to lie. You could show the code for an analysis of one set of data, but then show the plot or results from a different set or maybe a subset with outliers removed. *Do not do this.*

## 3.3 HIDE SPECIFIC OUTPUT

Sometimes, you want to display and run the code and you want to see some output, but you want to hide things like error messages, warnings, or other messages. You can do that with `error = FALSE`, `warning = FALSE`, and `message = FALSE`, respectively.

For example, if you want to load the `tidyverse` package, there are some messages that are displayed when you do. These just show what other packages are being installed and what conflicts there are.

```
library(tidyverse)
```

```
## ── Attaching packages ─────────────────────────────────── tidyverse 1.2.1 ──

## ✓ ggplot2 2.2.1     ✓ purrr   0.2.4
## ✓ tibble  1.3.4     ✓ dplyr   0.7.4
## ✓ tidyr   0.7.2     ✓ stringr 1.2.0
## ✓ readr   1.1.1     ✓ forcats 0.2.0

## ── Conflicts ────────────────────────────────────── tidyverse_conflicts() ──
## × dplyr::filter() masks stats::filter()
## × dplyr::lag()    masks stats::lag()
```

You almost never want to include this in your final document. Since these are neither error nor warning messages and are just regular FYI messages (just because they're red doesn't mean they're wrong), we can hide them with `message = FALSE`. So this code…

```
```{r, message = FALSE}
library(tidyverse)
```
```

…gives you this result…

```
library(tidyverse)
```

So hiding the messages is a great way to make sure your final document looks as good as it can.

### *3.3.1 Your turn!*

**The challenge**

1. When you make a histogram in ggplot2, unless you supply your own binwidth, it'll choose one for you and will let you know. Your task is to make a histogram and purposely leave the `binwidth` argument off to trigger the message. But instead of supplying a binwidth to suppress the message, just hide it using the code block options.

2. When you use `read_csv` to load data into R, it'll give you a message saying how each column was parsed like this:

```
    read_csv("http://joeystanley.com/data/cereal.csv")
```


orcid.org/0000-0002-9185-0048

```
    ## Parsed with column specification:
    ## cols(
    ##   name = col_character(),
    ##   mfr = col_character(),
    ##   type = col_character(),
    ##   shelf = col_integer(),
    ##   weight = col_double(),
    ##   cups = col_double(),
    ##   rating = col_double(),
    ##   calories = col_integer(),
    ##   sugars = col_integer(),
    ##   protein = col_integer(),
    ##   fat = col_integer(),
    ##   sodium = col_integer(),
    ##   fiber = col_double()
    ## )
```

So you can copy and paste that into the code itself like this:

```
    read_csv("http://joeystanley.com/data/cereal.csv",
             col_types = cols(
                 name = col_character(),
                 mfr = col_character(),
                 type = col_character(),
                 shelf = col_integer(),
                 weight = col_double(),
                 cups = col_double(),
                 rating = col_double(),
                 calories = col_integer(),
                 sugars = col_integer(),
                 protein = col_integer(),
                 fat = col_integer(),
                 sodium = col_integer(),
                 fiber = col_double()
             ))
```

But that also makes your code super long and probably isn't necessary to include in a final document. It's a lose-lose. Your task is to find a way to read your data in using `read_csv` without all those columns appearing in the final document.

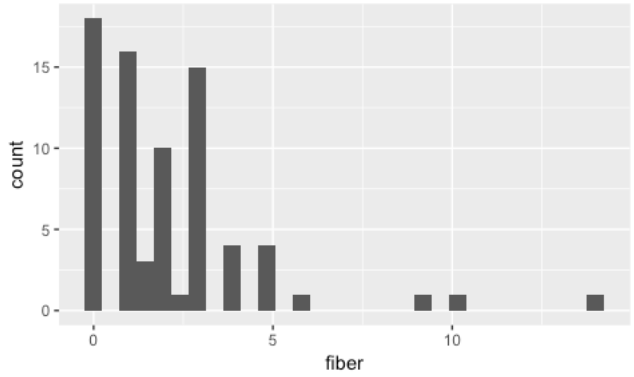
#### The solution

I'll make a histogram of the fiber in all the cereals. Here's the code with the warning message included.

````
```{r}
ggplot(cereal, aes(fiber)) +
    geom_histogram()
```

ggplot(cereal, aes(fiber)) +
    geom_histogram()

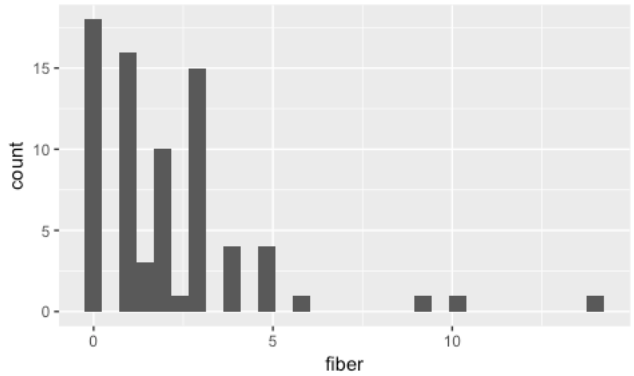
## `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
````




And here's the code with the message hidden:

````
```{r, message = FALSE}
ggplot(cereal, aes(fiber)) +
    geom_histogram()
```

ggplot(cereal, aes(fiber)) +
    geom_histogram()
````




For reading data in, the simple solution would be to hide the message:

````
```{r, message = FALSE}
read_csv("http://joeystanley.com/data/cereal.csv")
```
````

An alternative method—especially if you actually do want to modify some of those column types—is to use two blocks of code: one to display and the other to run.

````
```{r, eval = FALSE}
read_csv("http://joeystanley.com/data/cereal.csv")
```
```{r, echo = FALSE}
read_csv("http://joeystanley.com/data/cereal.csv",
         col_types = cols(
             name = col_character(),
             mfr = col_factor(levels = NULL), # modified
````


orcid.org/0000-0002-9185-0048
````

```
            type = col_factor(levels = NULL), # modified
            shelf = col_integer(),
            weight = col_double(),
            cups = col_double(),
            rating = col_double(),
            calories = col_integer(),
            sugars = col_integer(),
            protein = col_integer(),
            fat = col_integer(),
            sodium = col_integer(),
            fiber = col_double()
        ))
```

## 3.4 Figure options

There are lots of other options for code blocks for more advanced users. But some of them have to do with how figures are displayed. You can control many aspects of your figure using one or more options.

### *3.4.1 Size and quality*

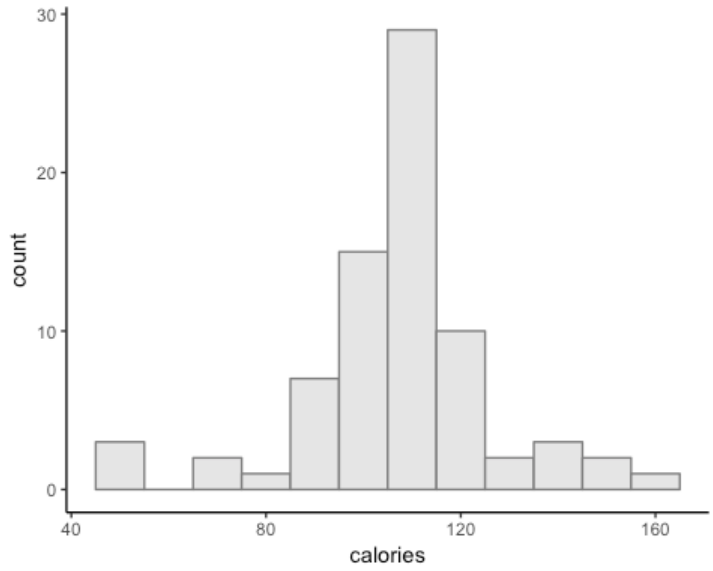
By default, the figure's height and width is 7 inches. This is fine for most purposes, but I like to make mine shorter to save on paper if this is going to be a pdf or if I'm displaying a histogram. (I actually did this already in the histograms displayed in this document!) The width is good, but one time I made a barplot with a couple dozen bars and it made sense to just turn it sideways and make the plot very tall. The default quality is 72 dpi (= dots per inch), but I like to set mine to the publication-standard 300. It makes the rendering and display a little bit slower so I usually wait until I'm close to finishing the document, but it makes the figures a lot crisper looking.

The way you can modify these settings is with the `fig.height`, `fig.width`, and `dpi` options. Let's take a basic histogram of the amount of calories per serving in the cereal data and change these options.

```
plot_calories <- ggplot(cereal, aes(x = calories)) +
    geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
    theme_classic()
```

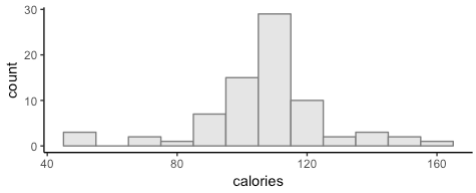
Here are the default settings.

```
```{r}
plot_calories
```
```



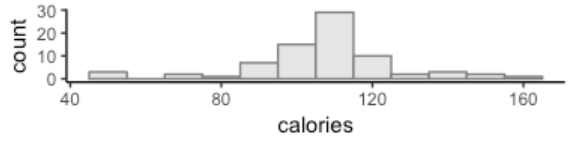

Here, I've just made it shorter with `fig.height = 2`.

```
```{r, fig.height = 2}
plot_calories
```
```




I've made it shorter using `fig.height = 1` and narrower using `fig.height = 4`.

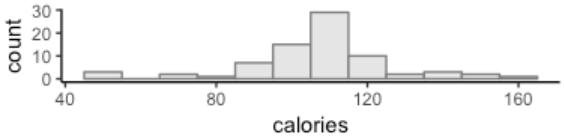
```
```{r, fig.height = 1, fig.width = 4}
plot_calories
```
```




For these narrower ones, you might want to also add `fig.align = "center"` if you want to stop it from being left-aligned.

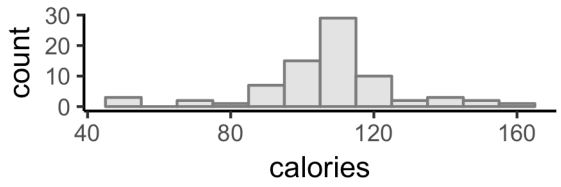
```
```{r, fig.height = 1, fig.width = 4, fig.align = "center"}
plot_calories
```
```


orcid.org/0000-0002-9185-0048




With `dpi = 300`, the resolution is increased so the plot itself will get bigger. You may have to scale the plot size down a bit more if you increase the resolution.

```
```{r, fig.height = 1, fig.width = 3, dpi = 300}
plot_calories
```
```




### *3.4.2 Your turn!*

Make any plot you want of any part of the cereal or menu datasets. Feel free to spruce up the plot a bit by adding color (as we learned in the ggplot2 workshops) or by creating new columns like calories per cup (like we learned in the tidyverse workshop). Display the plot in your RMarkdown file and make it an appropriate size and dimension.

# 4 OUTPUT

Everything we've done so far has used the default output settings. But we can modify how your document is rendered to make the HTML, Word, or PDF file look better.

## 4.1 DIFFERENT OUTPUT FILES

We take care of this all the way at the top of your RMarkdown file in the header. This is the portion between the `---` lines, where your title, name, and date appear. We can add lots more to that to make everything look good.

Most of these options come after the `output:` line, usually indented. In our document we have `html_document` next, but if you want to also create a Word file, you can add it to the heading like this:

```
output:
  html_document: default
  word_document: default
```

When you knit your document up, you'll end up with two new files in the same folder as your RMarkdown file. Since `html_document` is first, an HTML file will pop up (and if you put the `word_document` first, a Word file will open automatically). An HTML file is one that is what you might see on a webpage, and in fact you can open the file in a web browser and it'll look like a regular website (except it's just a file saved on your computer).

You can also create a PDF using `pdf_document: default` and it will create a PDF version of your file. This uses something called LaTeX, so if you're familiar with that there are lots of additional options that make the transition between R and LaTeX smoother. However, it might not work if you don't have it installed on your computer. That's okay: you can just knit it to a Word file and then save it as a PDF file (which is what I do).

### *4.1.1 Your turn!*

Try knitting your file into a Word file and look at how it renders. What kinds of changes do you see?

## 4.2 Table of contents

When you knit your files to an HTML or Word file, there are lots of defaults. You can change some of these defaults to make your document turn out better. To do this, you'll have to modify your header to make room for these new options by removing the `default` from the line, and adding new lines like this:

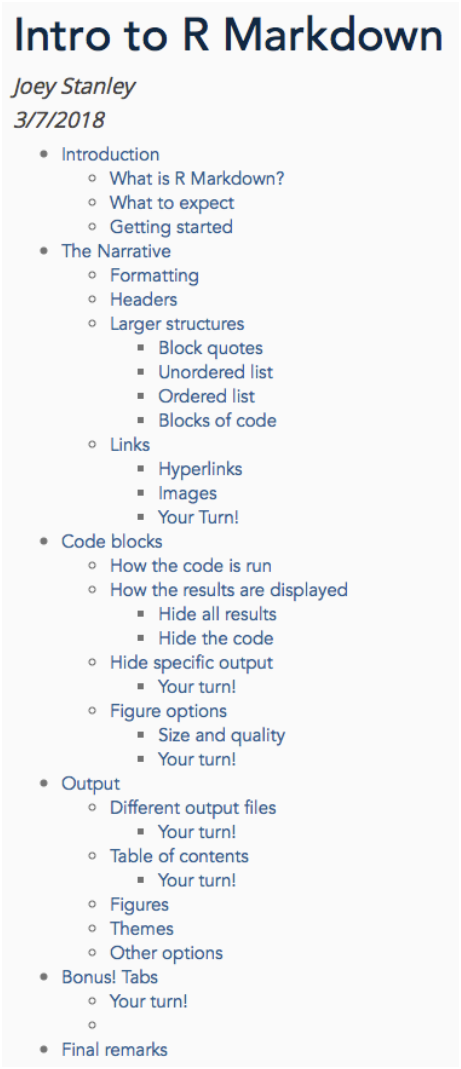
```
output:
    html_document:
        (Options go here.)
    word_document:
        (Other options go here.)
```

One of the options I like to add is a table of contents. This is something that shows up at the top of the document, which we can add this with `toc: true`.

```
output:
    html_document:
        toc: true
    word_document:
        toc: true
```

When you knit your files, you should now see a table of contents at the top. In Word, this is just like one that you might add using the controls in Word. In the HTML file, you'll see a list of links at the top.


orcid.org/0000-0002-9185-0048




In my opinion, the HTML one is not that pretty, so I like to add `toc_float: true`, which will move it to the side

```
output:
    html_document:
        toc: true
        toc_float: true
```

This is also super handy because it sort of floats along there at the top of the window even if you scroll down. If I were you, always put a floating table of contents if you have a table contents at all.

Depending on how many headings and subheadings you have, you might want to control how they look. In HTML files, you can make it so that the table of contents only shows Header 1 and Header 2 and anything deeper than that is hidden. You can do that with the option `toc_depth: 2`. This is what I typically do.

```
output:
    html_document:
        toc: true
        toc_float: true
        toc_depth: 2
```

Of course, if you don't use headers that often, then this won't matter much.


One last thing for the table of contents is to add a bit of flare and make other sections collapse down when you're not in them. Again, this is something that I like to do. This is actually an option within `toc_float`, so take out the `true`, add a colon, and on the next line, put `collapsed: true`:

```
output:
    html_document:
        toc: true
        toc_float:
            collapsed: true
        toc_depth: 2
```

You can also add `smooth_scoll = false` if you don't like the smooth transitions between sections, but I think they're kinda nice.

### *4.2.1 Your turn!*

Add a table of contents to your RMarkdown file.

## 4.3 Figures

If you include lots of figures, you might find yourself settings the same options over and over for each one. So if you're creating a Word file and the plots are just too big, you might want to scale them back a bit to save space.

Turns out you can override the default figure options so that you don't have to do a manual override in every code block. You can do this by using the `fig_width` and `fig_height` options in your header, either for the HTML output or the Word file.

```
output:
    html_document:
        toc: true
        toc_float:
            collapsed: true
        toc_depth: 2
        fig_width: 4
        fig_height: 4
    word_document:
        toc: true
        fig_width: 4
        fig_height: 4
```

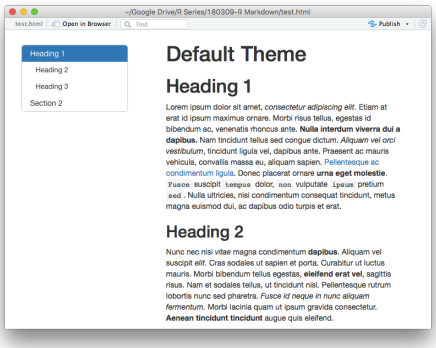
## 4.4 Themes

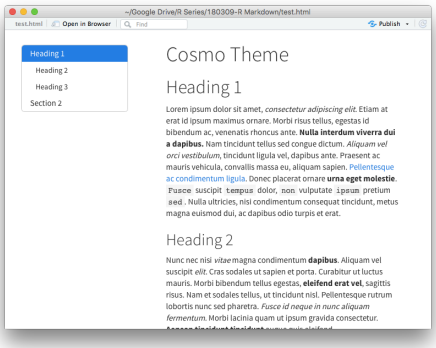
If you don't like the way the HTML file looks, you can actually change the theme of the file using the `theme:` option.

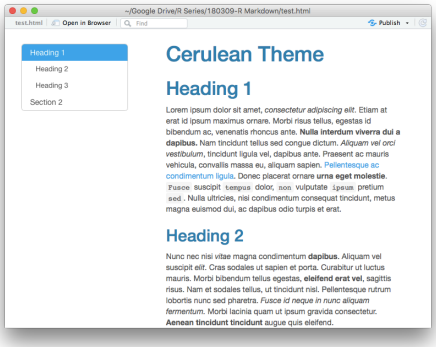
```
output:
    html_document:
        theme: "journal"
```

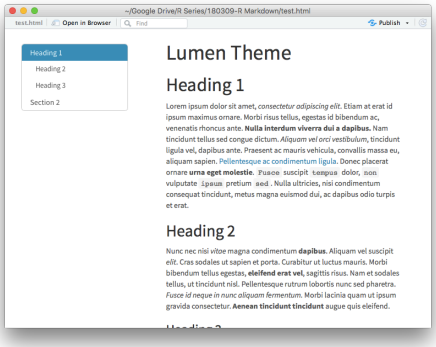
The choices you have are these:

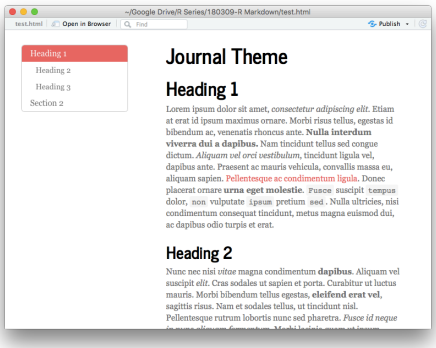

orcid.org/0000-0002-9185-0048

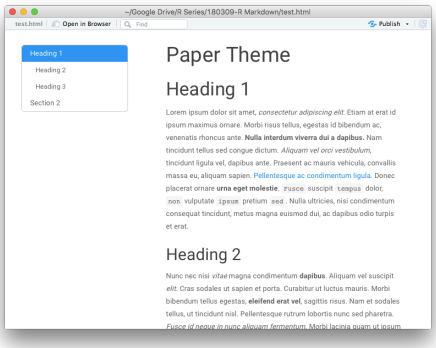

Default Theme
Heading 1
Heading 2
Heading 3
Section 2

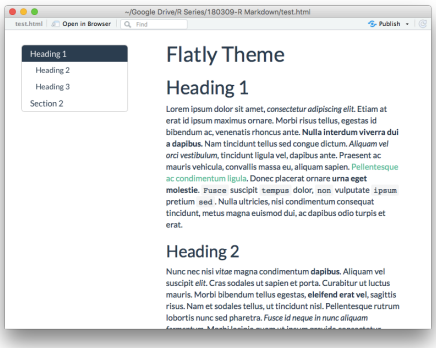

Cosmo Theme
Heading 1
Heading 2
Heading 3
Section 2

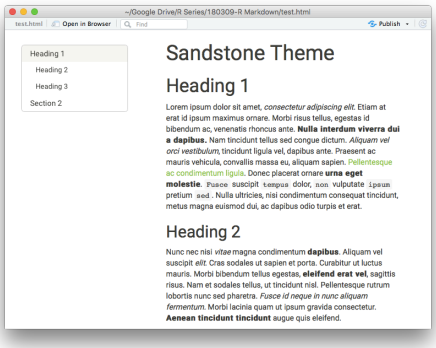

Cerulean Theme
Heading 1
Heading 2
Heading 3
Section 2

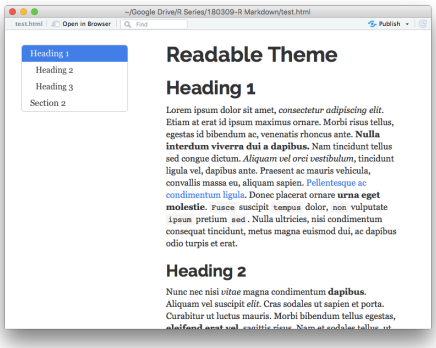

Lumen Theme
Heading 1
Heading 2
Heading 3
Section 2

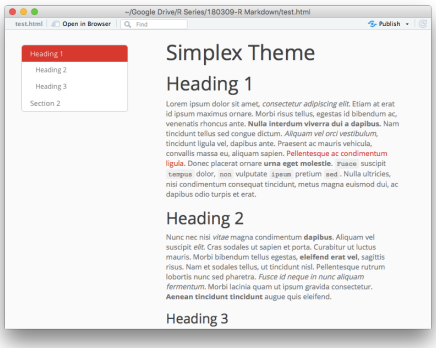

Journal Theme
Heading 1
Heading 2
Heading 3
Section 2

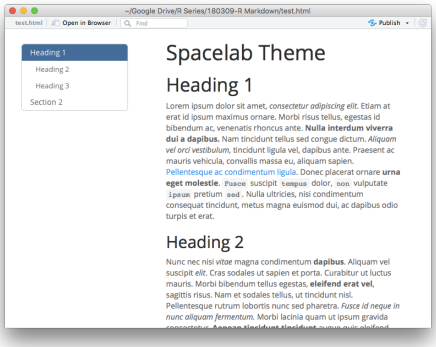

Paper Theme
Heading 1
Heading 2
Heading 3
Section 2

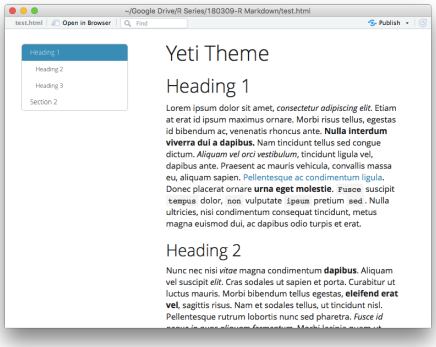

Flatly Theme
Heading 1
Heading 2
Heading 3
Section 2

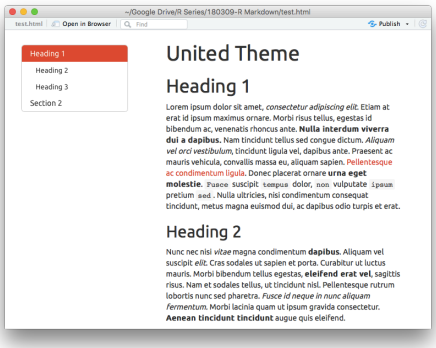

Sandstone Theme
Heading 1
Heading 2
Heading 3
Section 2




You can also just change the color of the syntax highlighting with `highlight`.

```
output:
    html_document:
        theme: "journal"
        highlight: "pygments"
```

Your options are these:


orcid.org/0000-0002-9185-0048


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Default Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Espresso Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Tango Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Zenburn Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Pygments Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Haddock Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Kate Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Textmate Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
Monochrome Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()


~/Google Drive/R Series/180309-R Markdown/code_samples.html
code_samples.html
Open in Browser
Find
Publish
NULL Highlight
library(ggplot2)
# Read in data from my website
cereal <- read.csv("http://joeystanley.com/data/cereal.csv")
# Plot it!
ggplot(cereal, aes(x = calories)) +
geom_histogram(binwidth = 10, color = "grey50", fill = "grey90") +
theme_classic()

If you happen to know CSS, you can also override the themes entirely and provide your own. I do this actually so that my RMarkdown files match my website. All you need to do is provide the path to the CSS file:

```
output:
  html_document:
    css: r-series.css
```

Of course, if you have no idea what CSS is, that's totally fine. You can stick with one of the default themes and you'll be perfectly alright.

### 4.5 Other options

There are *lots* and lots of other options that you can add to your file. Some more advanced than others. Some that might be useful to you include these:

- If you need to show tables, `df_print: paged` makes them render a lot prettier.
- Sometimes, you don't want to show the code by default, but you want to still give users the option to see it. The `code_folding: hide` in an HTML document makes it so that all the code is hidden by default, but there's a little button where each one would be that lets you see the code. It also adds one to the top to show or hide all code blocks.
- RMarkdown can automatically number the sections for you, which is pretty cool. This is nice because you don't ever have to worry about them, and you can rearrange sections all you want and it'll take care of making sure all the numbers are in the right place. You can add this with `number_sections: true`.

I encourage you to look through the RMarkdown documentation online[4] to find more options that you can use in your code. It's quite thorough and gives very clear examples.

## 5 Final remarks

RMarkdown is great. I use it all the time now because they're just a great way to incorporate prose into your R code. Today's workshop covered how to add text formatting to the narrative portion, how to modify the code blocks, and how to set global option for the output. It doesn't take too much time to turn your code into a nice document and you can learn about the nitty-gritty options as you go along—I learned a few things just by preparing this document! This takes some practice, but you'll get the hang of it the more you use it.

---

[4] https://rmarkdown.rstudio.com/html_document_format.html


orcid.org/0000-0002-9185-0048