Example APA7 Manuscript Made with Quarto and apaquarto

Natalie Dowling

MA Program in the Social Sciences, University of Chicago

Author Note

This example is based on the apaquarto template, created by W. Joel Schneider as part of the apaquarto extension for the Quarto publishing system. The author is grateful Dr. Marisa Casillas for creating the original version of this course. Author roles were classified using the Contributor Role Taxonomy (CRediT; https://credit.niso.org/) as follows: Natalie Dowling: conceptualization, writing

Correspondence concerning this article should be addressed to Natalie Dowling, MA Program in the Social Sciences, University of Chicago, 1155 E 60th St., Chicago, IL 60637, USA, Email: ndowling@uchicago.edu

Abstract

This document is a template demonstrating the apaquarto format. It includes examples of how to create figures and tables, as well as how to reference them in the text. The document is written in Quarto, a system for creating documents with R Markdown. The apaquarto extension provides a template for creating APA7-formatted manuscripts.

Keywords: R programming, ggplot2, data communication

Example APA7 Manuscript Made with Quarto and apaquarto

This document is a template demonstrating the apaquarto format. It includes examples of how to create figures and tables, as well as how to reference them in the text. The document is written in Quarto, a system for creating documents with R Markdown. The apaquarto extension provides a template for creating APA7-formatted manuscripts.

When rendered into html, pdf, or Word, this example produces an APA styled manuscript. Although the contents of the manuscript are not what you would expect in a psychology journal article, the formatting should demonstrate both the capabilities of the apaquarto extension and the basic template an actual manuscript would follow.

You can learn more about APA style in the APA Style Manual. Details about creating documents using the apaquarto extension in the documentation.

The example demonstrates the mechanics of markdown manuscripts using Quarto and the apaquarto extension, specifically. While many of the topics covered are the same in other markdown systems, like the older R Markdown or the `papaja` package, the specific syntax and options may differ.

Literature Review

Methods

Tables

In this section, we will go over how to create, render, and reference tables in apaquarto documents.

For tables produced by executable code cells, include a label with a `tbl-` prefix to make them cross-referenceable. For example:

In the chunk above, I produce Table 1 a table of the first five rows of the `iris` dataset. It renders from the chunk (it is not saved to an object) at the location of the chunk in the manuscript. I can reference this table in the text as `@tbl-iris`, which will render as “Table 1” in the rendered document.

Table 1, when rendered, has three components:

Table 1

Iris Data

Sepal.Length	Sepal.Width	Petal.Length	Petal.Width	Species
5.1	3.5	1.4	0.2	setosa
4.9	3.0	1.4	0.2	setosa
4.7	3.2	1.3	0.2	setosa
4.6	3.1	1.5	0.2	setosa
5.0	3.6	1.4	0.2	setosa
5.4	3.9	1.7	0.4	setosa

1. The table itself
2. The table *label*, which is the word “Table” and the generated number in the order it was rendered
3. The table caption, which is the title of the table

Take note that the `tbl-cap` option that assigned the “caption” is actually assigning with APA7 would refer to as the “title” of the table.

There are several important things to note about the simple figure chunk above:

1. The chunk options are in the “comment style” format. They are preceded by `#|` and are within the chunk, not the chunk header (the “‘`{r}` part).
2. The first chunk option, label, is in the line *immediately following the chunk header*. This is important for the table to be recognized as a table. If there is anything above the label, including comments and whitespace, the table will not render as a table and the document may not render at all without error
3. The label begins with “tbl-”, which tells apaquarto that this is a table. If the chunk has any other name, it will still render images in the chunk (including generated plots), but they will not be recognized as tables.

4. The caption in the `tbl-cap` option. As discussed above, this "caption" is the title of the table and should be in title case, with no period.

Results

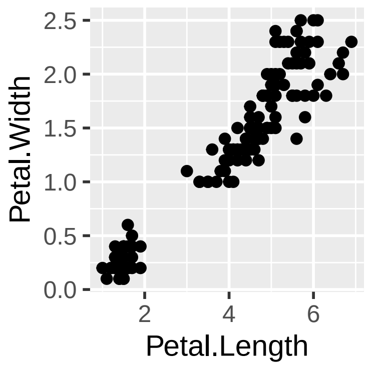
Figures can be created in R chunks and rendered as figures in the text. They can be referenced in the text and will be numbered in the order they are rendered.

Figure chunk options

Figure chunks should use the quarto-preferred "comment style" chunk option settings. Minimally, they should include a label and a caption. The label should begin with "fig-" to be recognized as a figure. The caption should be a string in title case. The image (file or object) should be rendered in the chunk.

Figure 1

The "Caption" (aka Title) of a Rendered Plot



In the chunk above, I produce Figure 1 a scatterplot of the `iris` dataset. It renders from the chunk (it is not saved to an object) at the location of the chunk in the manuscript. I can reference this plot in the text as `@fig-iris-rendered-plot`, which will render as "Figure 1" in the rendered document.

Figure 1, when rendered, has three components:

1. The plot itself

2. The figure *label*, which is the word “Figure” and the generated number in the order it was rendered
3. The figure caption, which is the title of the plot

Take note that the `fig-cap` option that assigned the “caption” is actually assigning with APA7 would refer to as the “title” of the figure. The thing you probably think of when you hear caption – the explanatory text below the figure – is called a “note.” There is an option to include a note, but it is not required or included in this minimal example.

There are several important things to note about the simple figure chunk above:

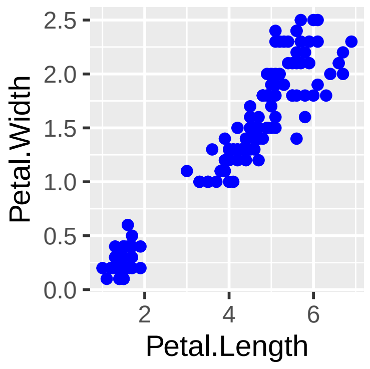
1. The chunk options are in the “comment style” format. They are preceded by `#|` and are within the chunk, not the chunk header (the “‘`{r}` part).
2. The first chunk option, label, is in the line *immediately following the chunk header*. This is important for the figure to be recognized as a figure. If there is anything above the label, including comments and whitespace, the figure will not render as a figure and the document may not render at all without error
3. The label begins with “fig-”, which tells apaquarto that this is a figure. If the chunk has any other name, it will still render images in the chunk (including generated plots), but they will not be recognized as figures.
4. The caption in the `fig-cap` option. As discussed above, this “caption” is the title of the figure and should be in title case, with no period.

Figure 1 is rendered by actually creating a ggplot in the chunk. However, you can also create a ggplot object elsewhere and render it as a figure. This is useful if you want to use the same plot in multiple places in the document, or if you want to create a very complicated plot in a sourced script.

Creating a chunk that assigns a plot to an object does not render the plot, since R code that assigns an object does not return anything. It only creates the object `iris_plot`. To render the plot, you need to call the object in a chunk with the figure options.

Figure 2

The Caption of a Rendered Plot Object



I can render the plot in a separate chunk by calling the plot object `iris_plot` in the chunk. This will render the plot as a figure in the text. I can reference this plot as Figure 2.

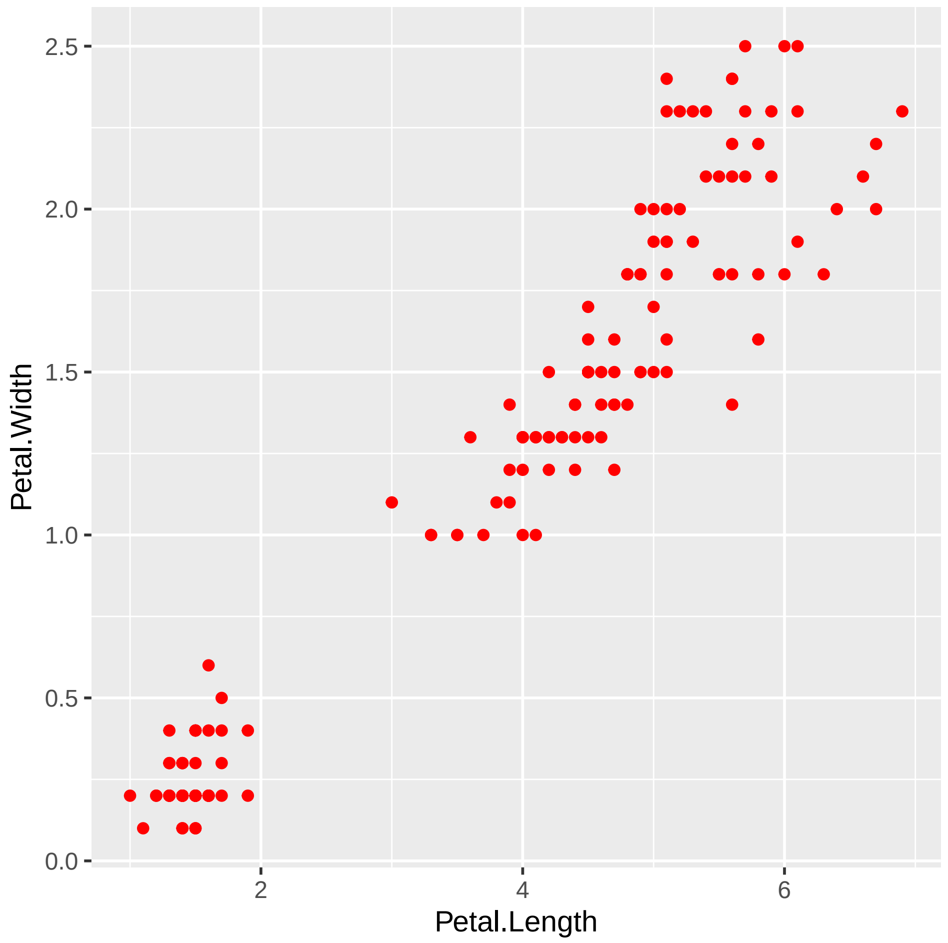
Quarto, and the apaquarto extension, can accept many additional chunk options. Like the label and caption options, these are set in the comment style within the chunk. Here are some additional options that can be used in figure chunks:

1. `apa-note`: A note that appears below the figure caption. This is the explanatory text that you might think of as a "caption" in other contexts. Unlike the options that follow, this is a feature of apaquarto specifically, not markdown or Quarto.
2. `fig-scap`: A short caption that appears in the list of figures. This is useful for long figure captions that are unwieldy in a list of figures.
3. `fig-alt`: Alt text for accessibility. This will appear if you render as HTML and is useful for screen readers.
4. `fig-align`: The alignment of the figure. This can be "left", "right", or "center" (the default).
5. `fig-width` and `fig-height`: The width and height of the figure in inches. This is useful for controlling the size of the figure in the rendered document. The default is 7 inches wide and 5 inches tall.

You can see all the effects of these chunk options in Figure 3. The figure is aligned to the

Figure 3

A Plot With More Chunk Options



Note. A note appearing below the figure.

right, has a width of 5 inches and a height of 6 inches, and has a note below the caption. The short caption is what will appear in the list of figures, and the alt text will appear when a reader mouses over the figure in an HTML document (it does not do anything in pdf or Word documents).

Referencing figures. As seen in the text above, figures can be referenced in text using the `@` symbol followed by the figure label. This will render as "Figure X" in the rendered document, where X is the order in which the figure was rendered. The figure label should be unique and begin with "fig-" to be recognized as a figure.

Plots are the only thing that count as figures. In APA documents, all images are typically treated as figures. You can include images as figures in your document by rendering them in a chunk using the `include_graphics()` function from the knitr package. Like with any other figure, the chunk required a label beginning with "fig-" and a caption and may take additional chunk options.

For images, it's usually better to use `out-width` or `out-height` rather than `fig-width` and `fig-height`. The image files already have an inherent size (unlike rendered plots), which can create problems when you try to give them new absolute dimensions (with the `fig-` options). The `out-` options let you use a relative sizing as a percentage, which is usually more reliable.

Figure 4

Types of Scientific Papers

TYPES OF SCIENTIFIC PAPER
WE PUT A CAMERA SOMEWHERE NEW
HEY, I FOUND A TROVE OF OLD RECORDS! THEY DON'T TURN OUT TO BE PARTICULARLY USEFUL, BUT STILL, COOL!
MY COLLEAGUE IS WRONG AND I CAN FINALLY PROVE IT
THE IMMUNE SYSTEM IS AT IT AGAIN
WE FIGURED OUT HOW TO MAKE THIS EXOTIC MATERIAL, SO EMAIL US IF YOU NEED SOME
WHAT ARE FISH EVEN DOING DOWN THERE
THIS TASK I HAD TO DO ANYWAY TURNED OUT TO BE HARD ENOUGH FOR ITS OWN PAPER
HEY, AT LEAST WE SHOWED THAT THIS METHOD CAN PRODUCE RESULTS! THAT'S NOT NOTHING, RIGHT?
CHECK OUT THIS WEIRD THING ONE OF US SAW WHILE OUT FOR A WALK
WE ARE 500 SCIENTISTS AND HERE'S WHAT WE'VE BEEN UP TO FOR THE LAST 10 YEARS
SOME THOUGHTS ON HOW EVERYONE ELSE IS BAD AT RESEARCH
WE SCANNED SOME UNDERGRADUATES

Note. From xkcd (#2456) by Randall Monroe

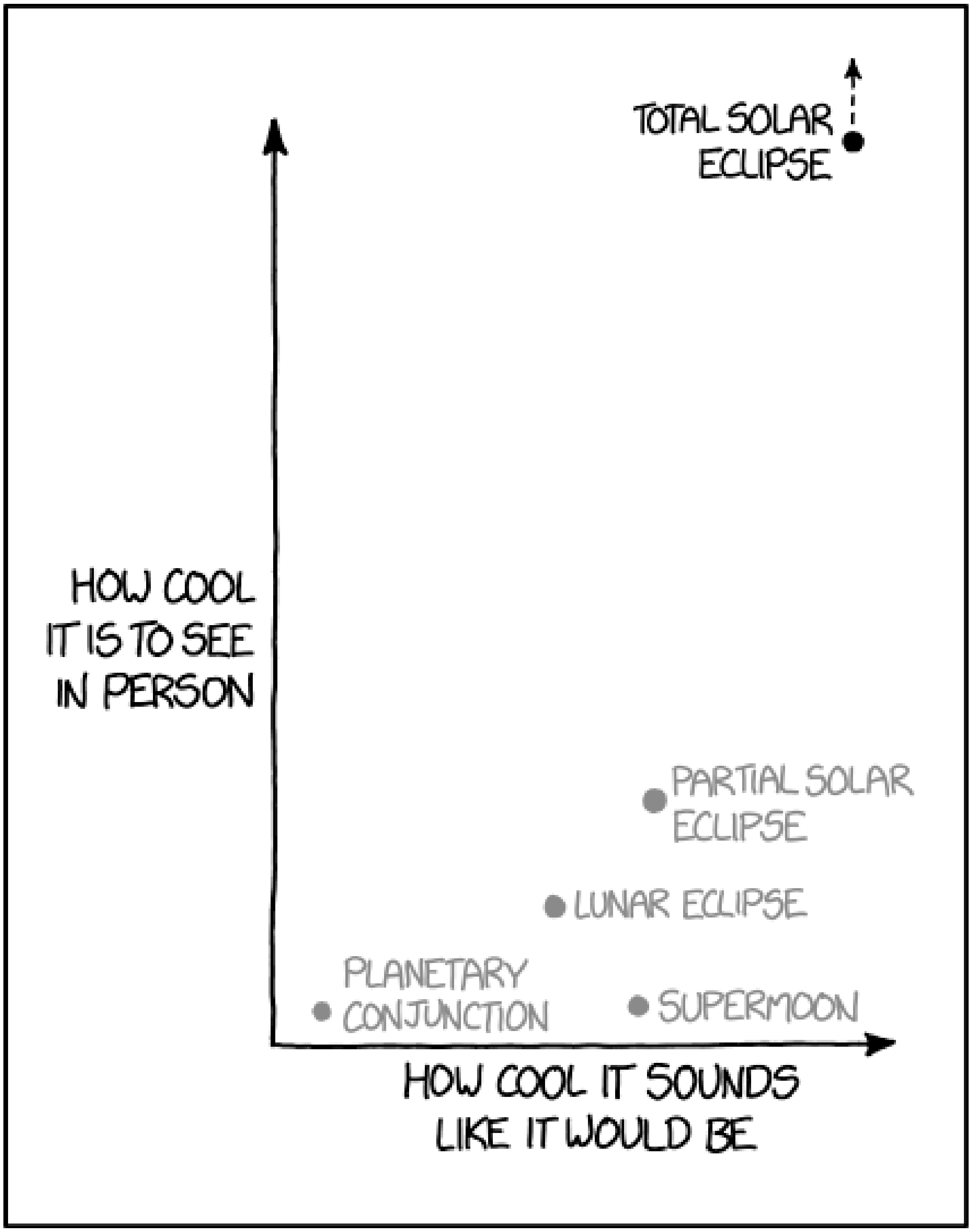
Why not take the opportunity to look at a few more xkcd comics?

When you knit this document to a pdf, knitr/Quarto will try to pick the best location to include it. If your image is very large, or if there is very little text between images, the location it chooses may not be precisely where you put the chunk. You can force the image to render exactly where the chunk is with the option `fig-pos: "H"`. Either way–the default or the forced hold–can produce unexpected consequences, so keep an eye out for issues and try out alternatives as needed.

Back to xkcd. Here's a good one about types of eclipses:

Figure 5

Types of Eclipses

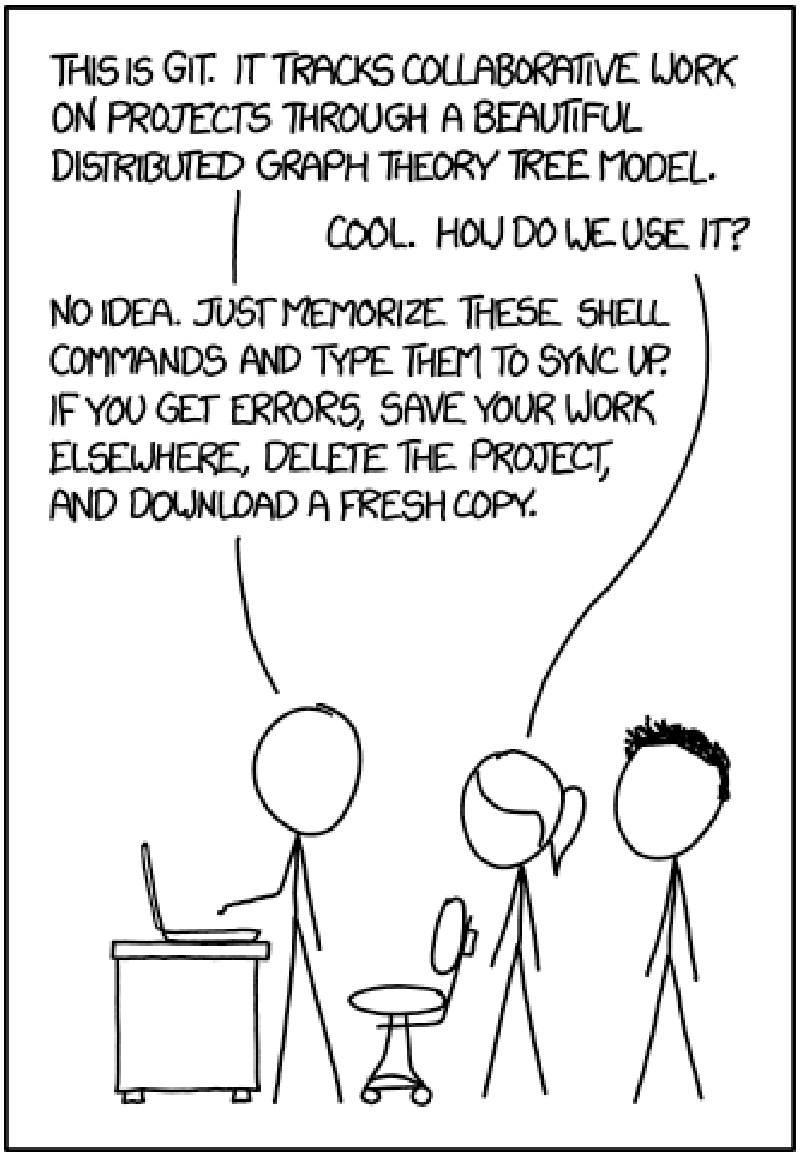


Note. From xkcd (#1880) by Randall Monroe

And of course there's this classic about literally everyone's experience using git:

Figure 6

The Tragedy of git


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Note. From xkcd (#1597) by Randall Monroe

Love a good xkcd. They're all great, but right now Figure 6 really speaks to me. (You too, maybe?) Figure 5 is good and it's got a plot, but it's not as relevant to our class as Figure 4 or Figure 6.

I have a point. Notice that the numbers assigned to each figure are based on the order in which they are rendered, not the order in which they are referenced.

Analyses & Inline Code

In this section, we discuss running and referencing statistical analyses.

A critical component of using our D2MR workflow for reproducible research is the ability to seamlessly integrate references to objects and object elements into the narrative text of your publication. When you run a model or summary and then identify coefficients, summary stats, or other important values in the output, you *could* type out those values as literal, static text. This is not only time-consuming, it problematically opens the door for significant human error each time the values must be manually updated.

Inline R code is an alternative that allows you to replace this static text with dynamic code. Rather than changing the β, p-value, mean, etc. every time any tiny thing in your data or code processes change, the values are calculated during the knitting process. You add the code once, then (ideally) never need to modify it so long as the structure of your data and modeling stays the same.

Fundamentally, think about inline R code as a teeny-tiny code chunk that takes the "1-chunk-1-thing" rule to the extreme, so much so that the output of the whole thing is the same kind of thing you'd just type into a Word document otherwise. Everything within the code will run exactly like it's run in the console, meaning you can always quickly check how it will render when knit by just copying and pasting that code into your console in RStudio.

The general format for inline R is a backtick[1], then a space, then as much code as you like (remember, literally what you'd see if you copied and pasted it into the console), then a closing backtick:

```
`r yourCodeHere()`
```

[1] If you want to make something appear as "verbatim" text that looks like code the way it shows up in this notebook, surround it in backticks without the "r" in front: `` `verbatim text` ``. If you want to include backticks within the literal code, smash your head against your desk until you give up.

Referencing values from lists and tibbles. In the following chunk, I create a vector, a list, and a tibble. These are basic R objects you'll need to extract references from often.

To reference a value from a vector, list, or tibble, you need to know the index of the value you want to reference. In R, indexing starts at 1[2], so the first element of a vector, list, or tibble is at index 1. You can reference the value at a specific index by using the object name followed by square brackets containing the index. For example, to reference the second element of a vector `ex_vector`, you would write `ex_vector[2]`: 30.

If you've got a list that is really a list and not a vector that you're casually calling a "list," it's a little weirder. List objects look like a series of values, but they are actually a series of other lists. If you referenced the second element of a list object `ex_list` with `ex_list[2]`, you would get a list that contains the second element: 30.[3] To get the actual value, you need to reference the index of the value within the list. For example, to reference the second element of a list object `ex_list`, you would write `ex_list[[2]]`: 30.

You can reference values from a tibble in the same way you would reference values from a vector, but you have to index both the row and column. For example, to reference the value in the second row and third column of a tibble `ex_tibble`, you would write `ex_tibble[2, 3]`: 75.

You can also use brackets to index by names. In a tibble, that usually means column/variable names, but you could use row names if your df has them. For example, to reference the value in the second row and the column named "mass" in a tibble `ex_tibble`, you would write `ex_tibble[2, "mass"]` to get the same as the above `ex_tibble[2,3]`: 75.

In your code, you'll often need to reference whole columns or rows, which you can do by omitting a value on either side of the comma. So to get the second row of `ex_tibble`, you would

[2] This is probably the biggest reason I will never be a python person. I'm team counting-starts-at-1 all the way. It's the *1st* thing! There's a 1 right in the name!

[3] Quarto is actually smart enough to know you're trying to reference the value and not the list, so if you use this in your markdown it will knit to the value. But if you're actually using it for any kind of programming (not just printing directly), you'll need to use the correct data type.

write `ex_tibble[2, ]`: C-3PO, 167, 75, list(c("A New Hope", "The Empire Strikes Back", "Return of the Jedi", "The Phantom Menace", "Attack of the Clones", "Revenge of the Sith")).

You *can* call a tibble *row* in markdown and render it as text.

To get the column named "mass" from `ex_tibble`, you would write `ex_tibble[, "mass"]`. If you don't include the comma, it will assume you want the variable. So `ex_tibble["mass"]` will return the column named "mass". More commonly, if you want to call a column or row, you can use the `$` operator. For example, to reference the column named "mass" from `ex_tibble`, you would write `ex_tibble$mass`.

You *cannot* call a tibble *column* in markdown and render it as text. (What would that even look like?)

Issues. Something knitting in a way you didn't expect? Check that the class of your output is what you think it should be. In particular, if you want something to appear as regular-old in-line text, it needs to be a *value* data type, like numeric, character, or logical. The `cor()` function, for example, results in a simple numeric value, while the `cor.test()` function results in an "htest" object. That object contains numeric values you can extract and reference, but if you try to reference it directly it won't knit.