Test Quarto Book

Brad Cannell

2023-06-27

Table of contents

Preface **3**
Useful websites: . . . 3
Rendering . . . 3
Publishing to GitHub pages . . . 4

I part_my_chapters.qmd **5**

1 Book Options **6**

2 Other little things **7**
2.1 Pros and cons . . . 7

II Authoring **8**

3 Rmd Documents **9**
3.1 Let's Get Programming . . . 9
3.2 Simulating data . . . 9
3.3 Vectors . . . 10
3.3.1 Vector types . . . 12
3.3.2 Double vectors . . . 12
3.3.3 Integer vectors . . . 12
3.3.4 Logical vectors . . . 13
3.4 Data frames . . . 14
3.5 Tibbles . . . 16
3.5.1 The as_tibble function . . . 17
3.5.2 The tibble function . . . 18
3.5.3 The tribble function . . . 19
3.5.4 Why use tibbles . . . 20
3.6 Missing data . . . 22
3.7 Our first analysis . . . 24
3.7.1 Manual calculation of the mean . . . 24
3.7.2 Dollar sign notation . . . 25
3.7.3 Bracket notation . . . 25
3.7.4 The sum function . . . 26

3.7.5 Nesting functions . . . 27
3.7.6 The length function . . . 28
3.7.7 The mean function . . . 30
3.8 Some common errors . . . 30
3.9 Summary . . . 32

4 Images **33**
4.1 Native Quarto figures . . . 33
4.2 Adding figures with Knitr . . . 34

5 Gifs **36**

6 Cross References **39**
6.1 Book parts . . . 39
6.2 Citations . . . 39

7 Adding call out boxes **40**

III Working with Code **42**

8 Code chunks **43**
8.1 Code Run Time . . . 43
8.2 Naming code chunks . . . 43
8.3 Showing entire code chunks . . . 43

9 Sourcing qmd files **45**

IV Publishing **46**

10 PDF **47**

11 Publishing **48**
11.1 GitHub repository . . . 48
11.2 GitHub Pages . . . 48
11.3 Netlify . . . 49

V Built-in Chapters **50**

References **51**

Appendices **52**

A Example appendix **52**

Preface

This is my first Quarto book. For now, I'm just using it for some experimentation. Eventually, I will probably want to add all of this to a GitHub repository. Right now, Olivia is in class and I'm content with just experimenting a little bit.

This is **not** my new R Notes book; although, I may move R Notes over to Quarto at some point. This is purely just a sandbox for playing with Quarto books.

Useful websites:

- Quarto book documentation

Rendering

You can render the files by clicking the Render button in RStudio. To render the HTML and PDF files at the same time, type `quarto render` into the terminal. You can also render Quarto files with a native R code chunk.

- The input argument: The input file or project directory to be rendered (defaults to rendering the project in the current working directory).
- The output_format argument: Target output format (defaults to "html"). The option "all" will render all formats defined within the file or project.

```
```{r}
#| Render with R
#| eval: false
quarto::quarto_render(output_format = "all")
```
```

Publishing to GitHub pages

This article is great. After committing, and making sure you are on the main branch, type `quarto publish gh-pages` in the terminal.

Part I

part_my_chapters.qmd

1 Book Options

A collection of notes on Quarto book options.

- Link to list of book options
- How do you add a cover image? Look at r4ds.
  - `cover-image: cover.jpg`
- How do you add last date rendered to the `_quarto.yml` file? "r Sys.Date()" doesn't seem to work.
  - Use the keyword today
- Can I add links to GitHub and/or social media?
  - Still need an answer here.
- Can I add links to the GitHub repo containing the books files?
  - Yes. See https://quarto.org/docs/books/book-output.html#sidebar-tools
- How do I add a favicon?
  - `favicon: cover.jpg`
- How do I add Google analytics?
  - Still need an answer here.
- Can I add a Google analytics badge to my GitHub README?
  - Still need an answer here.
- How do I preview the book in my web browser instead of RStudio's Viewer pane?
  - Just click the little gear icon next to the `Render` button in RStudio. Select `Preview in Window`.

2 Other little things

2.1 Pros and cons

Some of the things I like about working with Quarto (as opposed to bookdown) so far

- A preview of the book renders automatically.
- I can easily render only one chapter by opening that chapter's qmd file and clicking the Render button.

Part II

Authoring

3 Rmd Documents

I copied this Rmd document over from the Bookdown version of R4Epi (05_lets_get_programming.Rmd). Just trying to figure out what happens when we mix and mingle Rmd and qmd documents.

It seems to work well!

3.1 Let's Get Programming

In this chapter, we are going to tie together many of the concepts we've learned so far, and you are going to create your first basic R program. Specifically, you are going to write a program that simulates some data and analyzes it.

3.2 Simulating data

Data simulation can be really complicated, but it doesn't have to be. It is simply the process of *creating* data as opposed to *finding data in the wild*. This can be really useful in several different ways.

1. Simulating data is really useful for getting help with a problem you are trying to solve. Often, it isn't feasible for you to send other people the actual data set you are working on when you encounter a problem you need help with. Sometimes, it may not even be legally allowed (i.e., for privacy reasons). Instead of sending them your entire data set, you can simulate a little data set that recreates the challenge you are trying to address without all the other complexity of the full data set. As a bonus, I have often found that I end up figuring out the solution to the problem I'm trying to solve as I recreate the problem in a simulated data set that I intended to share with others.
2. Simulated data can also be useful for learning about and testing statistical assumptions. In epidemiology, we use statistics to draw conclusions about populations of people we are interested in based on samples of people drawn from the population. Because we don't actually have data from *all* the people in the population, we have to make some assumptions about the population based on what we find in our sample. When we simulate data, we know the truth about our population because we *created* our population to have that truth. We can then use this simulated population to play "what if" games

with our analysis. *What if we only sampled half as many people? What if their heights aren't actually normally distributed? What if we used a probit model instead of a logit model?* Going through this process and answering these questions can help us understand how much, and under what circumstances, we can trust the answers we found in the real world.

So, let's go ahead and write a complete R program to simulate and analyze some data. As I said, it doesn't have to be complicated. In fact, in just a few lines of R code below we simulate and analyze some data about a hypothetical class.

```
class <- data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, 72)
)
```

```
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

```
mean(class$heights)
```

```
[1] 68.5
```

As you can see, this data frame contains the students' names and heights. We also use the `mean()` function to calculate the average height of the class. By the end of this chapter, you will understand all the elements of this R code and how to simulate your own data.

3.3 Vectors

Vectors are the most fundamental data structure in R. Here, data structure means "container for our data." There are other data structures as well; however, they are all built from vectors. That's why I say vectors are the most fundamental data structure. Some of these other structures include matrices, lists, and data frames. In this book, we won't use matrices or lists much at all, so you can forget about them for now. Instead, we will almost exclusively use data frames to hold and manipulate our data. However, because data frames are built

from vectors, it can be useful to start by learning a little bit about them. Let's create our first vector now.

```
# Create an example vector
names <- c("John", "Sally", "Brad", "Anne")
# Print contents to the screen
names
```

```
[1] "John"  "Sally" "Brad"  "Anne"
```

Here's what we did above:

- We *created* a vector of names with the `c()` (short for combine) function.
  - The vector contains four values: "John", "Sally", "Brad", and "Anne".
  - All of the values are character strings (i.e., words). We know this because all of the values are wrapped with quotation marks.
  - Here we used double quotes above, but we could have also used single quotes. We cannot, however, mix double and single quotes for each character string. For example, `c("John', ...)` won't work.
- We *assigned* that vector of character strings to the word `names` using the `<-` function.
  - R now recognizes `names` as an **object** that we can do things with.
  - R programmers may refer to the names object as "the names object", "the names vector", or "the names variable". For our purposes, these all mean the same thing.
- We *printed* the contents of the `names` object to the screen by typing the word "names".
  - R **returns** (shows us) the four character values ("John" "Sally" "Brad" "Anne") on the computer screen.

Try copying and pasting the code above into the RStudio console on your computer. You should notice the names vector appear in your **global environment**. You may also notice that the global environment pane gives you some additional information about this vector to the right of its name. Specifically, you should see `chr [1:4] "John"  "Sally" "Brad" "Anne"`. This is R telling us that `names` is a character vector (`chr`), with four values (`[1:4]`), and the first four values are `"John"  "Sally" "Brad"  "Anne"`.

3.3.1 Vector types

There are several different vector **types**, but each vector can have only one type. The type of the vector above was character. We can validate that with the `typeof()` function like so:

```
typeof(names)
```

```
[1] "character"
```

The other vector types that we will use in this book are double, integer, and logical. Double vectors hold real numbers and integer vectors hold integers. Collectively, double vectors and integer vectors are known as numeric vectors. Logical vectors can only hold the values TRUE and FALSE. Here are some examples of each:

3.3.2 Double vectors

```
# A numeric vector
my_numbers <- c(12.5, 13.98765, pi)
my_numbers
```

```
[1] 12.500000 13.987650  3.141593
```

```
typeof(my_numbers)
```

```
[1] "double"
```

3.3.3 Integer vectors

Creating integer vectors involves a weird little quirk of the R language. For some reason, and I have no idea why, we must type an "L" behind the number to make it an integer.

```
# An integer vector - first attempt
my_ints_1 <- c(1, 2, 3)
my_ints_1
```

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

```
typeof(my_ints_1)
```

```
[1] "double"
```

```
# An integer vector - second attempt
# Must put "L" behind the number to make it an integer. No idea why they chose "L".
my_ints_2 <- c(1L, 2L, 3L)
my_ints_2
```

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

```
typeof(my_ints_2)
```

```
[1] "integer"
```

3.3.4 Logical vectors

```
# A logical vector
# Type TRUE and FALSE in all caps
my_logical <- c(TRUE, FALSE, TRUE)
my_logical
```

```
[1]  TRUE FALSE  TRUE
```

```
typeof(my_logical)
```

```
[1] "logical"
```

Rather than have an abstract discussion about the particulars of each of these vector types right now, I think it's best to wait and learn more about them when they naturally arise in the context of a real challenge we are trying to solve with data. At this point, just having some vague idea that they exist is good enough.

3.4 Data frames

Vectors are useful for storing a single characteristic where all the data is of the same type. However, in epidemiology, we typically want to store information about many different characteristics of whatever we happen to be studying. For example, we didn't just want the names of the people in our class, we also wanted the heights. Of course, we can also store the heights in a vector like so:

```
heights <- c(68, 63, 71, 72)
heights
```

```
[1] 68 63 71 72
```

But this vector, in and of itself, doesn't tell us which height goes with which person. When we want to create relationships between our vectors, we can use them to build a data frame. For example:

```
# Create a vector of names
names <- c("John", "Sally", "Brad", "Anne")
# Create a vector of heights
heights <- c(68, 63, 71, 72)
# Combine them into a data frame
class <- data.frame(names, heights)
# Print the data frame to the screen
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

Here's what we did above:

- We *created* a data frame with the `data.frame()` function.
  - The first argument we passed to the `data.frame()` function was a vector of names that we previously created.
  - The second argument we passed to the `data.frame()` function was a vector of heights that we previously created.

- We *assigned* that data frame to the word `class` using the `<-` function.
  - R now recognizes `class` as an **object** that we can do things with.
  - R programmers may refer to this class object as "the class object" or "the class data frame". For our purposes, these all mean the same thing. We could also call it a data set, but that term isn't used much in R circles.
- We *printed* the contents of the `class` object to the screen by typing the word "class".
  - R **returns** (shows us) the data frame on the computer screen.

Try copying and pasting the code above into the RStudio console on your computer. You should notice the `class` data frame appear in your **global environment**. You may also notice that the global environment pane gives you some additional information about this data frame to the right of its name. Specifically, you should see `4 obs. of 2 variables`. This is R telling us that `class` has four rows or observations (`4 obs.`) and two columns or variables (`2 variables`). If you click the little blue arrow to the left of the data frame's name, you will see information about the individual vectors that make up the data frame.

As a shortcut, instead of creating individual vectors and then combining them into a data frame as we've done above, most R programmers will create the vectors (columns) directly inside of the data frame function like this:

```
# Create the class data frame
class <- data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, 72)
) # Closing parenthesis down here.

# Print the data frame to the screen
class
```

```
  names heights
1  John      68
2 Sally      63
3  Brad      71
4  Anne      72
```

As you can see, both methods produce the exact same result. The second method, however, requires a little less typing and results in fewer objects cluttering up your global environment. What I mean by that is that the `names` and `heights` vectors won't exist independently in your global environment. Rather, they will only exist as columns of the `class` data frame.

You may have also noticed that when we created the `names` and `heights` vectors (columns) directly inside of the `data.frame()` function we used the equal sign (`=`) to assign values instead of the assignment arrow (`<-`). This is just one of those quirky R exceptions we talked about in the chapter on speaking R's language. In fact, `=` and `<-` can be used interchangeably in R. It is only by convention that we usually use `<-` for assigning values, but use `=` for assigning values to columns in data frames. I don't know why this is the convention. If it were up to me, we wouldn't do this. We would just pick `=` or `<-` and use it in all cases where we want to assign values. But, it isn't up to me and I gave up on trying to fight it a long time ago. Your R programming life will be easier if you just learn to assign values this way – even if it's dumb.

Warning: By definition, all columns in a data frame must have the same length (i.e., number of rows). That means that each vector you create when building your data frame must have the same number of values in it. For example, the class data frame above has four names and four heights. If we had only entered three heights, we would have gotten the following error: `Error in data.frame(names = c("John", "Sally", "Brad", "Anne"), heights = c(68,  : arguments imply differing number of rows: 4, 3`

3.5 Tibbles

Tibbles are a data structure that come from another tidyverse package – the `tibble` package. Tibbles *are* data frames and serve the same purpose in R that data frames serve; however, they are enhanced in several ways. You are welcome to look over the tibble documentation or the tibbles chapter in R for Data Science if you are interested in learning about all the differences between tibbles and data frames. For our purposes, there are really only a couple things I want you to know about tibbles right now.

First, tibbles are a part of the `tibble` package – NOT base R. Therefore, we have to install and load either the `tibble` package or the `dplyr` package (which loads the tibble package for us behind the scenes) before we can create tibbles. I typically just load the `dplyr` package.

```
# Install the dplyr package. YOU ONLY NEED TO DO THIS ONE TIME.
install.packages("dplyr")
```

```
# Load the dplyr package. YOU NEED TO DO THIS EVERY TIME YOU START A NEW R SESSION.
library(dplyr)
```

Second, we can create tibbles using one of three functions: `as_tibble()`, `tibble()`, or `tribble()`. I'll show you some examples shortly.

Third, try not to be confused by the terminology. Remember, tibbles *are* data frames. They are just enhanced data frames.

3.5.1 The as_tibble function

We use the `as_tibble()` function to turn an already existing basic data frame into a tibble. For example:

```
# Create a data frame
my_df <- data.frame(
  name = c("john", "alexis", "Steph", "Quiera"),
  age  = c(24, 44, 26, 25)
)

# Print my_df to the screen
my_df
```

```
    name age
1   john  24
2 alexis  44
3  Steph  26
4 Quiera  25
```

```
# View the class of my_df
class(my_df)
```

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

Here's what we did above:

- We used the `data.frame()` function to create a new data frame called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is a data frame.

```
# Use as_tibble() to turn my_df into a tibble
my_df <- as_tibble(my_df)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
3 Steph     26
4 Quiera    25
```

```r
# View the class of my_df
class(my_df)
```

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

Here's what we did above:

- We used the `as_tibble()` function to turn `my_df` into a tibble.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.

3.5.2 The tibble function

We can use the `tibble()` function in place of the `data.frame()` function when we want to create a tibble from scratch. For example:

```r
# Create a data frame
my_df <- tibble(
  name = c("john", "alexis", "Steph", "Quiera"),
  age  = c(24, 44, 26, 25)
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
```

```
3 Steph     26
4 Quiera    25
```

```r
# View the class of my_df
class(my_df)
```

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

Here's what we did above:

- We used the `tibble()` function to create a new tibble called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.

3.5.3 The tribble function

Alternatively, we can use the `tribble()` function in place of the `data.frame()` function when we want to create a tibble from scratch. For example:

```r
# Create a data frame
my_df <- tribble(
  ~name,    ~age,
  "john",   24,
  "alexis", 44,
  "Steph",  26,
  "Quiera", 25
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 2
  name     age
  <chr>  <dbl>
1 john      24
2 alexis    44
3 Steph     26
4 Quiera    25
```

```
# View the class of my_df
class(my_df)
```

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

Here's what we did above:

- We used the `tribble()` function to create a new tibble called `my_df`.
- We used the `class()` function to view `my_df`'s class (i.e., what kind of object it is).
  - The result returned by the `class()` function tells us that `my_df` is still a data frame, but it is also a tibble. That's what "tbl_df" and "tbl" mean.
- There is absolutely no difference between the tibble we created above with the `tibble()` function and the tibble we created above with the `tribble()` function. The only difference between the two functions is the syntax we used to pass the column names and data values to each function.
  - When we use the `tibble()` function, we pass the data values to the function horizontally as vectors. This is the same syntax that the `data.frame()` function expects us to use.
  - When we use the `tribble()` function, we pass the data values to the function vertically instead. The only reason this function exists is because it can sometimes be more convenient to type in our data values this way. That's it.
  - Remember to type a tilde ("~") in front of your column names when using the `tribble()` function. For example, type `~name` instead of `name`. That's how R knows you're giving it a column name instead of a data value.

3.5.4 Why use tibbles

At this point, some students wonder, "If tibbles are just data frames, why use them? Why not just use the `data.frame()` function?" That's a fair question. As I have said multiple times already, tibbles are enhanced. However, I don't believe that going into detail about those enhancements is going to be useful to most of you at this point – and may even be confusing. But, I will show you one quick example that's pretty self-explanatory.

Let's say that we are given some data that contains four people's age in years. We want to create a data frame from that data. However, let's say that we also want a column in our new data frame that contains those same ages in months. Well, we could do the math ourselves. We could just multiply each age in years by 12 (for the sake of simplicity, assume that everyone's age in years is gathered on their birthday). But, we'd rather have R do the math for us. We

can do so by asking R to multiply each value of the the column called `age_years` by 12. Take a look:

```
# Create a data frame using the data.frame() function
my_df <- data.frame(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25),
  age_months = age_years * 12
)
```

```
Error in data.frame(name = c("john", "alexis", "Steph", "Quiera"), age_years = c(24, : objec
```

Uh, oh! We got an error! This error says that the column `age_years` can't be found. How can that be? We are clearly passing the column name `age_years` to the `data.frame()` function in the code chunk above. Unfortunately, the `data.frame()` function doesn't allow us to *create* and *refer to* a column name in the same function call. So, we would need to break this task up into two steps if we wanted to use the `data.frame()` function. Here's one way we could do this:

```
# Create a data frame using the data.frame() function
my_df <- data.frame(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25)
)

# Add the age in months column to my_df
my_df <- my_df %>% mutate(age_months = age_years * 12)

# Print my_df to the screen
my_df
```

```
    name age_years age_months
1   john        24        288
2 alexis        44        528
3  Steph        26        312
4 Quiera        25        300
```

Alternatively, we can use the `tibble()` function to get the result we want in just one step like so:

```r
# Create a data frame using the tibble() function
my_df <- tibble(
  name       = c("john", "alexis", "Steph", "Quiera"),
  age_years  = c(24, 44, 26, 25),
  age_months = age_years * 12
)

# Print my_df to the screen
my_df
```

```
# A tibble: 4 x 3
  name   age_years age_months
  <chr>      <dbl>      <dbl>
1 john          24        288
2 alexis        44        528
3 Steph         26        312
4 Quiera        25        300
```

In summary, tibbles *are* data frames. For the most part, we will use the terms "tibble" and "data frame" interchangeably for the rest of the book. However, remember that tibbles are *enhanced* data frames. Therefore, there are some things that we will do with tibbles that we can't do with basic data frames.

3.6 Missing data

As indicated in the warning box at the end of the data frames section of this chapter, all columns in our data frames have to have the same length. So what do we do when we are truly missing information in some of our observations? For example, how do we create the `class` data frame if we are missing Anne's height for some reason?

In R, we represent missing data with an `NA`. For example:

```r
# Create the class data frame
data.frame(
  names   = c("John", "Sally", "Brad", "Anne"),
  heights = c(68, 63, 71, NA) # Now we are missing Anne's height
)
```

```
  names heights
1  John      68
```

```
2 Sally      63
3  Brad      71
4  Anne      NA
```

Warning: Make sure you capitalize `NA` and don't use any spaces or quotation marks. Also, make sure you use `NA` instead of writing `"Missing"` or something like that.

By default, R considers `NA` to be a logical-type value (as opposed to character or numeric). for example:

```
typeof(NA)
```

```
[1] "logical"
```

However, you can tell R to make `NA` a different type by using one of the more specific forms of `NA`. For example:

```
typeof(NA_character_)
```

```
[1] "character"
```

```
typeof(NA_integer_)
```

```
[1] "integer"
```

```
typeof(NA_real_)
```

```
[1] "double"
```

Most of the time, you won't have to worry about doing this because R will take care of converting `NA` for you. What do I mean by that? Well, remember that every vector can have only one type. So, when you add an `NA` (logical by default) to a vector with double values as we did above (i.e., `c(68, 63, 71, NA)`), that would cause you to have three double values and one logical value in the same vector, which is not allowed. Therefore, R will automatically convert the `NA` to `NA_real_` for you behind the scenes.

This is a concept known as "type coercion" and you can read more about it here if you are interested. As I said, most of the time you don't have to worry about type coercion – it will happen automatically. But, sometimes it doesn't and it will cause R to give you an error. I mostly encounter this when using the `if_else()` and `case_when()` functions, which we will discuss later.

3.7 Our first analysis

Congratulations on your new R programming skills. You can now create vectors and data frames. This is no small thing. Basically, everything else we do in this book will start with vectors and data frames.

Having said that, just *creating* data frames may not seem super exciting. So, let's round out this chapter with a basic descriptive analysis of the data we simulated. Specifically, let's find the average height of the class.

You will find that in R there are almost always many different ways to accomplish a given task. Sometimes, choosing one over another is simply a matter of preference. Other times, one method is clearly more efficient and/or accurate than another. This is a point that will come up over and over in this book. Let's use our desire to find the mean height of the class as an example.

3.7.1 Manual calculation of the mean

For starters, we can add up all the heights and divide by the total number of heights to find the mean.

```
(68 + 63 + 71 + 72) / 4
```

```
[1] 68.5
```

Here's what we did above:

- We used the addition operator (+) to add up all the heights.
- We used the division operator (/) to divide the sum of all the heights by 4 - the number of individual heights we added together.
- We used parentheses to enforce the correct order of operations (i.e., make R do addition before division).

This works, but why might it not be the best approach? Well, for starters, manually typing in the heights is error prone. We can easily accidently press the wrong key. Luckily, we already have the heights stored as a column in the `class` data frame. We can *access* or *refer to* a single column in a data frame using the **dollar sign notation**.

3.7.2 Dollar sign notation

```
class$heights
```

```
[1] 68 63 71 72
```

Here's what we did above:

- We used the dollar sign notation to *access* the `heights` column in the `class` data frame.
  - Dollar sign notation is just the data frame name, followed by the dollar sign, followed by the column name.

3.7.3 Bracket notation

Further, we can use **bracket notation** to access each value in a vector. I think it's easier to demonstrate bracket notation than it is to describe it. For example, we could access the third value in the names vector like this:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Bracket notation
# Access the third element in the heights vector with bracket notation
heights[3]
```

```
[1] 71
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and bracket notation, to access each individual value of the `height` column in the `class` data frame. This will help us get around the problem of typing each individual height value. For example:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
(class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4
```

```
[1] 68.5
```

3.7.4 The sum function

The second method is better in the sense that we no longer have to worry about mistyping the heights. However, who wants to type `class$heights[...]` over and over? What if we had a hundred numbers? What if we had a thousand numbers? This wouldn't work. Luckily, there is a function that adds all the numbers contained in a numeric vector – the `sum()` function. Let's take a look:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Add together all the individual heights with the sum function
sum(heights)
```

```
[1] 274
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and `sum()` function, to add up all the individual heights in the `heights` column of the `class` data frame. It looks like this:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4

# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
sum(class$heights) / 4
```

```
[1] 68.5
```

Here's what we did above:

- We passed the numeric vector `heights` from the `class` data frame to the `sum()` function using dollar sign notation.
- The `sum()` function returned the total value of all the heights added together.
- We divided the total value of the heights by four – the number of individual heights.

3.7.5 Nesting functions

!! Before we move on, I want to point out something that is actually kind of a big deal. In the third method above, we didn't manually add up all the individual heights - R did this calculation for us. Further, we didn't store the sum of the individual heights somewhere and then divide that stored value by 4. Heck, we didn't even see what the sum of the individual heights were. Instead, the returned value from the sum function (274) was used *directly* in the next calculation (`/ 4`) by R without us seeing the result. In other words, `(68 + 63 + 71 + 72) / 4`, `274 / 4`, and `sum(class$heights) / 4` are all exactly the same thing to R. However, the third method (`sum(class$heights) / 4`) is much more **scalable** (i.e., adding a lot more numbers doesn't make this any harder to do) and much less error prone. Just to be clear, the BIG DEAL is that we now know that the values returned by functions can be *directly* passed to other functions in exactly the same way as if we typed the values ourselves.

This concept, functions passing values to other functions is known as **nesting functions**. It's called nesting functions because we can put functions inside of other functions.

"But, Brad, there's only one function in the command `sum(class$heights) / 4` – the `sum()` function." Really? Is there? Remember when I said that operators are also functions in R? Well, the division operator is a function. And, like all functions it can be written with parentheses like this:

```
# Writing the division operator as a function with parentheses
`/`(8, 4)
```

```
[1] 2
```

Here's what we did above:

- We wrote the division operator in its more function-looking form.
  - Because the division operator isn't a letter, we had to wrap it in backticks (‘).
  - The backtick key is on the top left corner of your keyboard near the escape key (esc).
  - The first argument we passed to the division function was the dividend (The number we want to divide).
  - The second argument we passed to the division function was the divisor (The number we want to divide by).

So, the following two commands mean exactly the same thing to R:

```
8 / 4
```

```
`/`(8, 4)
```

And if we use this second form of the division operator, we can clearly see that one function is *nested* inside another function.

```
`/`(sum(class$heights), 4)
```

```
[1] 68.5
```

Here's what we did above:

- We calculated the mean height of the class.
  - The first argument we passed to the division function was the returned value from the `sum()` function.
  - The second argument we passed to the division function was the divisor (4).

This is kind of mind-blowing stuff the first time you encounter it. I wouldn't blame you if you are feeling overwhelmed or confused. The main points to take away from this section are:

1. Everything we *do* in R, we will *do* with functions. Even operators are functions, and they can be written in a form that looks function-like; however, we will almost never actually write them in that way.
2. Functions can be **nested**. This is huge because it allows us to directly pass returned values to other functions. Nesting functions in this way allows us to do very complex operations in a scalable way and without storing a bunch of unneeded values that are created in the intermediate steps of the operation.
3. The downside of nesting functions is that it can make our code difficult to read - especially when we nest many functions. Fortunately, we will learn to use the pipe operator (`%>%`) in the workflow basics part of this book. Once you get used to pipes, they will make nested functions much easier to read.

Now, let's get back to our analysis…

3.7.6 The length function

I think most of us would agree that the third method we learned for calculating the mean height is preferable to the first two methods for most situations. However, the third method still requires us to know how many individual heights are in the `heights` column (i.e., 4). Luckily, there is a function that tells us how many individual values are contained in a vector – the `length()` function. Let's take a look:

```
# Create the heights vector
heights <- c(68, 63, 71, 72)

# Return the number of individual values in heights
length(heights)
```

```
[1] 4
```

Remember, that data frame columns are also vectors. So, we can combine the dollar sign notation and `length()` function to automatically calculate the number of values in the `heights` column of the `class` data frame. It looks like this:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4

# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
# sum(class$heights) / 4

# Fourth way. Use dollar sign notation with the sum function and the length
# function
sum(class$heights) / length(class$heights)
```

```
[1] 68.5
```

Here's what we did above:

- We passed the numeric vector `heights` from the `class` data frame to the `sum()` function using dollar sign notation.
- The `sum()` function returned the total value of all the heights added together.
- We passed the numeric vector `heights` from the `class` data frame to the `length()` function using dollar sign notation.
- The `length()` function returned the total number of values in the `heights` column.
- We divided the total value of the heights by the total number of values in the `heights` column.

3.7.7 The mean function

The fourth method above is definitely the best method yet. However, this need to find the mean value of a numeric vector is so common that someone had the sense to create a function that takes care of all the above steps for us – the `mean()` function. And as you probably saw coming, we can use the mean function like so:

```
# First way to calculate the mean
# (68 + 63 + 71 + 72) / 4

# Second way. Use dollar sign notation and bracket notation so that we don't
# have to type individual heights
# (class$heights[1] + class$heights[2] + class$heights[3] + class$heights[4]) / 4

# Third way. Use dollar sign notation and sum function so that we don't have
# to type as much
# sum(class$heights) / 4

# Fourth way. Use dollar sign notation with the sum function and the length
# function
# sum(class$heights) / length(class$heights)

# Fifth way. Use dollar sign notation with the mean function
mean(class$heights)
```

```
[1] 68.5
```

Congratulations again! You completed your first analysis using R!

3.8 Some common errors

Before we move on, I want to briefly discuss a couple common errors that will frustrate many of you early in your R journey. You may have noticed that I went out of my way to differentiate between the `heights` vector and the `heights` column in the `class` data frame. As annoying as that may have been, I did it for a reason. The `heights` vector and the `heights` column in the `class` data frame are two separate things to the R interpreter, and you have to be very specific about which one you are referring to. To make this more concrete, let's add a `weight` column to our `class` data frame.

```
class$weight <- c(160, 170, 180, 190)
```

Here's what we did above:

- We created a new column in our data frame – `weight` – using dollar sign notation.

Now, let's find the mean weight of the students in our class.

```
mean(weight)
```

```
Error in mean(weight): object 'weight' not found
```

Uh, oh! What happened? Why is R saying that `weight` doesn't exist? We clearly created it above, right? Wrong. We didn't create an *object* called weight in the code chunk above. We created a *column* called `weight` in the *object* called `class` in the code chunk above. Those are *different things* to R. If we want to get the mean of `weight` we have to tell R that `weight` is a column in `class` like so:

```
mean(class$weight)
```

```
[1] 175
```

A related issue can arise when you have an object and a column with the same name but different values. For example:

```
# An object called scores
scores <- c(5, 9, 3)

# A columnmn in the class data frame called scores
class$scores <- c(95, 97, 93, 100)
```

If you ask R for the mean of `scores`, R will give you an answer.

```
mean(scores)
```

```
[1] 5.666667
```

However, if you wanted the mean of the `scores` column in the `class` data frame, this won't be the *correct* answer. Hopefully, you already know how to get the correct answer, which is:

```
mean(class$scores)
```

```
[1] 96.25
```

Again, the `scores` object and the `scores` column of the `class` object are different things to R.

3.9 Summary

Wow! We covered a lot in this first part of the book on getting started with R and RStudio. Don't feel bad if your head is swimming. It's a lot to take-in. However, you should feel proud of the fact that you can already do some legitimately useful things with R. Namely, simulate and analyze data. In the next part of this book, we are going to discuss some tools and best practices that will make it easier and more efficient for you to write and share your R code. After that, we will move on to tackling more advanced programming and data analysis challenges.

4 Images

Links

- The Quarto documentation about images
- Example image in R4DS

4.1 Native Quarto figures

Here are some examples of adding figures.

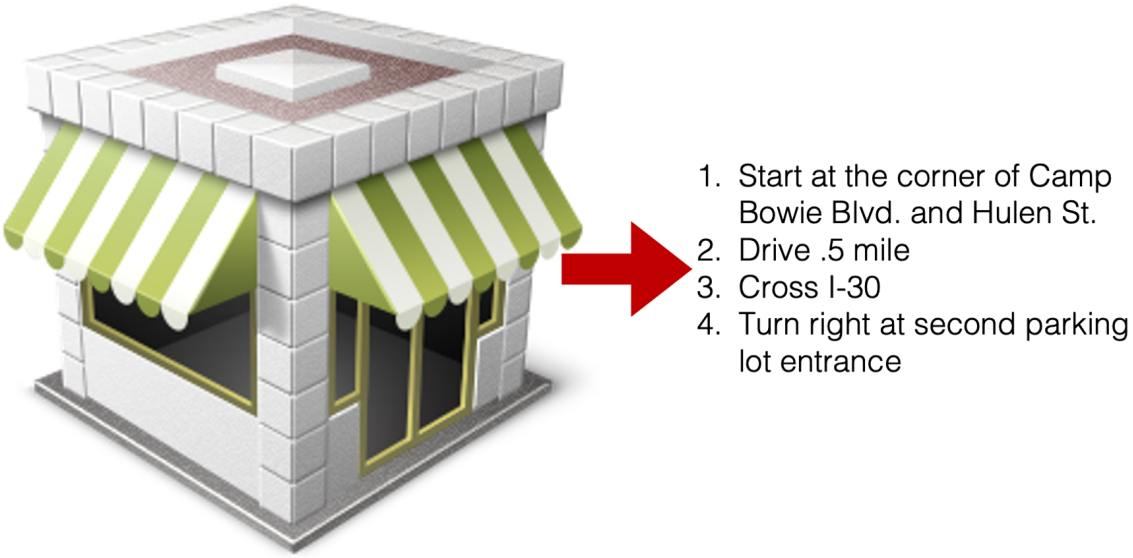

Figure 4.1: Relative Path Directions

And I can cross reference the figure by typing `@fig-directions`. See Figure 4.2

> **!** Important
>
> For cross-references to work, the image must have a caption *and* a label.

4.2 Adding figures with Knitr

In R4DS (link above), Hadley et al. are still using `knitr::include_graphics("path")` to insert images even though the book has been converted to Quarto documents. When using Bookdown, Yihui gives four arguments for using `knitr::include_graphics("path")` instead of native markdown image formatting. So, we will likely continue to use them too. Here is an example image code chunk from R4DS:

````
```{r}
#| label: fig-ds-diagram
#| echo: false
#| fig-cap: |
#|   In our model of the data science process, you start with data import
#|   and tidying. Next, you understand your data with an iterative cycle of
#|   transforming, visualizing, and modeling. You finish the process
#|   by communicating your results to other humans.
#| fig-alt: |
#|   A diagram displaying the data science cycle: Import -> Tidy -> Understand
#|   (which has the phases Transform -> Visualize -> Model in a cycle) ->
#|   Communicate. Surrounding all of these is Communicate.
#| out.width: NULL

knitr::include_graphics("diagrams/data-science/base.png", dpi = 270)
```
````

Here, I'm adding my own image with `knitr::include_graphics("path")`.

And I can cross reference the figure by typing `@fig-directions`. See Figure 4.2

> **!** Important
>
> For cross-references to work, the image must have a caption *and* a label.

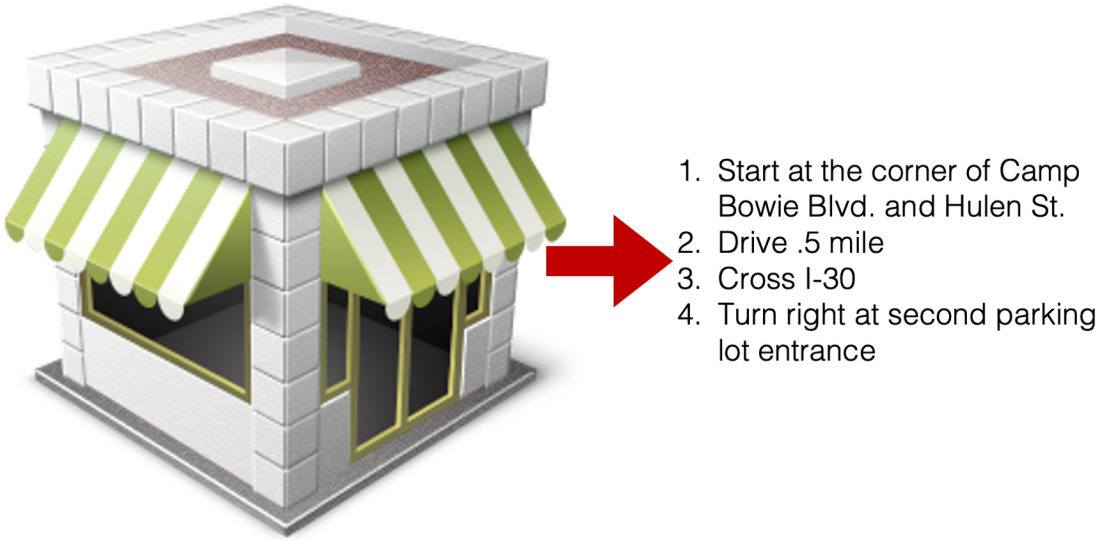

Figure 4.2: Relative Path Directions.

5 Gifs

In R4Epi, we use quite a few gifs. That makes rendering the book to pdf format challenging. What happens when we add a gif to a qmd document and render it to pdf?

The following code works great for HTML format, but it messes up pdf format.

```
![A gif about file paths](img/file_path_gif.gif){#fig-file-paths}

And I can [cross reference](https://quarto.org/docs/authoring/cross-references.html) the g

::: {.callout-important}
For cross-references to work, the image must have a caption _and_ a label.
:::
```

What if I use `knitr::include_graphics("path")`?

````
```{r}
#| label: fig-file-paths
#| echo: false
#| fig-cap: |
#|   A gif about file paths.
#| fig-alt: |
#|   A gif about file paths.

knitr::include_graphics("img/file_path_gif.gif")
```

And I can [cross reference](https://quarto.org/docs/authoring/cross-references.html) the f

::: {.callout-important}
For cross-references to work, the image must have a caption _and_ a label.
:::
````

No, this doesn't work either. Here is a pretty good discussion on the topic. It looks like the easiest route may be to use conditional content. Given the limited number of gifs in R4Epi, this shouldn't be too big of a problem.

So, start by conditionally displaying the gif if the output format is html.

````
::: {.content-visible when-format="html"}

```{r}
#| label: fig-file-paths
#| echo: false
#| fig-cap: |
#|   A gif about file paths.
#| fig-alt: |
#|   A gif about file paths.

knitr::include_graphics("img/file_path_gif.gif")
```


:::
````

Which renders as…

Then, conditionally add a thumbnail image of the gif when the output format is pdf.

- Create a duplicate of the gif in Finder.
- Add "_thumb" to the end of the file name (before the file extension).
- Open the duplicate file in preview.
- Use Preview's sidebar to keep only one of the thumbnail images from the gif (select and delete the rest).
- Click File > Export and export to png.
- Delete the duplicate gif.

````
::: {.content-visible when-format="pdf"}

```{r}
#| label: fig-file-paths
#| echo: false
#| fig-cap: |
#|   A thumbnail of gif about file paths.
#| fig-alt: |
#|   A thumbnail gif about file paths.

knitr::include_graphics("img/file_path_gif.gif")
```
````

```
:::
```

Which renders as...

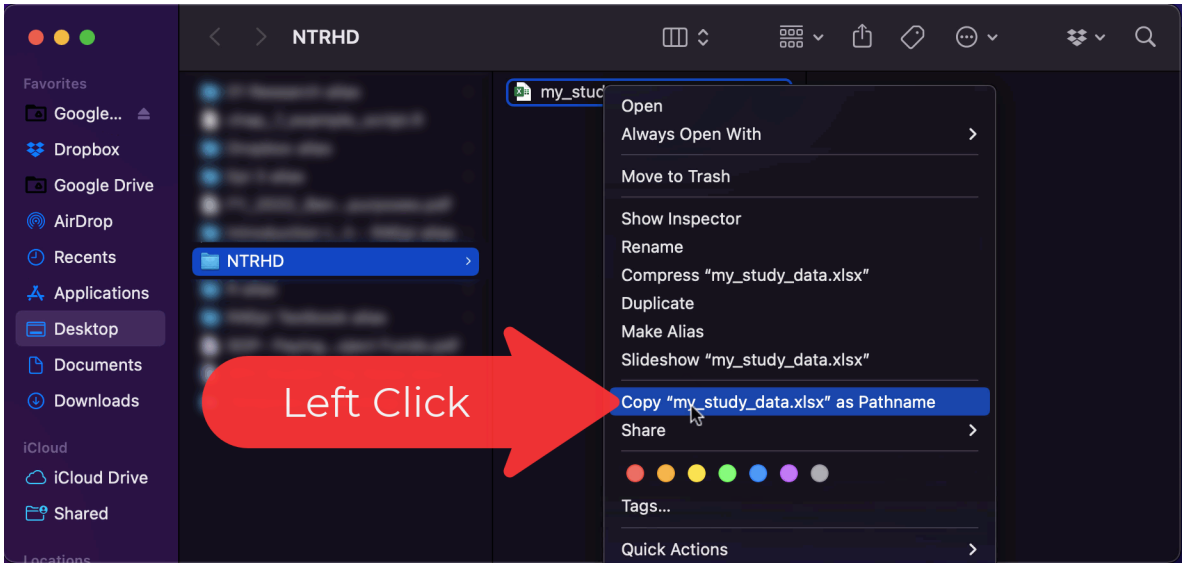

Figure 5.1: A thumbnail of gif about file paths.

6 Cross References

When authoring the book, it is common to need to cross reference book parts (i.e., chapters, sections), figures, tables, and I'm going to go ahead and include citations here too.

6.1 Book parts

Link to the official documentation

Gotcha's: - Sometimes I get errors like `WARNING: index.html: Unable to resolve crossref @sec-crossrefs`

6.2 Citations

Built-in example: See Knuth (1984) for additional discussion of literate programming.

7 Adding call out boxes

In R4Epi, we sometimes add special boxes for side notes and warnings. How do we add those into qmd books?

What do those boxes look like when we render the book to pdf format?

Does Quarto have some built-in boxes? It looks like it might?

- How do you add the call out boxes (i.e., important, etc.)?
  - See Quarto documentation on callout blocks

```
::: callout-note
Note that there are five types of callouts, including: `note`, `tip`, `warning`, `caution`
:::
```

> **Note**
>
> Note that there are five types of callouts, including: `note`, `tip`, `warning`, `caution`, and `important`.

```
::: callout-warning
Callouts provide a simple way to attract attention, for example, to this warning.
:::
```

> **Warning**
>
> Callouts provide a simple way to attract attention, for example, to this warning.

```
::: callout-important
## This is Important

Danger, callouts will really improve your writing.
:::
```

This is Important

Danger, callouts will really improve your writing.

```
::: callout-tip
## Tip With Title

This is an example of a callout with a title.
:::
```

Tip With Title

This is an example of a callout with a title.

```
::: {.callout-caution collapse="true"}
## Expand To Learn About Collapse

This is an example of a 'collapsed' caution callout that can be expanded by the user. You
:::
```

Expand To Learn About Collapse

This is an example of a ‘collapsed’ caution callout that can be expanded by the user. You can use `collapse="true"` to collapse it by default or `collapse="false"` to make a collapsible callout that is expanded by default.

Part III

Working with Code

8 Code chunks

These are my notes on working with code chunks.

Here are some other helpful resources:

- Need to add a link to a comprehensive guide

8.1 Code Run Time

We were interested in understanding how much caching code chunks could potentially speed up the rendering process for books. Initially, the code to test out caching code chunks was part of the `test_quarto_book` project. However, it seemed like the `quarto_render()` function works differently in the context of a book project and was making testing difficult. Specifically, `quarto::quarto_render("slow_code_default.qmd")` seemed to render the entire book instead of just the `slow_code_default.qmd` document. Therefore, we created a separate Quarto project to run these tests – `test_quarto_cached_chunks`.

The bottom line is that caching actually causes the render to take longer to process on the first run. However, it appears to speed up subsequent renders. This makes sense.

8.2 Naming code chunks

Use the `label` code chunk option. For example, `#| label: slow-code-cached`

8.3 Showing entire code chunks

When you want to display and entire code chunk in the rendered output, as opposed to executing the code chunk, surround it with four backticks.

Also, make sure to use double curly braces `{{}}` around the language name identifier. If you don't, funky looking code will be output.

````
````
```{r}
#| A code chunk for display
#| eval:false
1 + 1
```
````
````

9 Sourcing qmd files

We can "source" or programmatically render (as opposed to interactively rendering with the `Render` button) qmd files – including inside of code chunks in other qmd files. Here is an example code chunk.

```
quarto::quarto_render("slow_code_cached.qmd", quiet = TRUE)
```

Part IV

Publishing

10 PDF

Notes on exporting the book to pdf.

Not only how to render it to pdf, but is there an option

According to this, it looks like we can make a pdf downloadable by adding `downloads: [pdf, epub]` to `_quarto.yml`. In the config file for this test book, we added `downloads: pdf` to the book options section.

Important: It looks like the option above only creates the link on the HTML page to download the pdf version of the book. However, a new pdf version of the book isn't automatically rendered when we click the render button. You have to specifically render a pdf version separately from an html version.

We may want to think about creating an internal file with notes and a chunk that renders html and pdf when executed.

````
```{bash}
#| eval: false
quarto render
```
````

Even easier, you can do this with a native R code chunk.

- The input argument: The input file or project directory to be rendered (defaults to rendering the project in the current working directory).
- The output_format argument: Target output format (defaults to "html"). The option "all" will render all formats defined within the file or project.

````
```{r}
#| Render with R
#| eval: false
quarto::quarto_render(output_format = "all")
```
````

11 Publishing

This chapter is about getting the book files on GitHub and creating an HTML version of the book for public consumption. I'm starting with GitHub Pages because it seems like it should be the lowest hanging fruit.

11.1 GitHub repository

The first step to publishing a book online is to put it into a GitHub repository. Originally, I started my book project by creating a new project, clicking new directory, then Quarto book. After creating the book project, I created a repository for it in GitHub and then tried to use GitHub's on-screen instructions to push the book files to the repo. However, I kept getting errors and wasn't ever able to make it work. I feel like this shouldn't be the case and was probably just some weird one-off. So, don't give up on that process just yet.

What I eventually got to work was this process.

- I created the repo on GitHub (with nothing but a README file) first.
- I cloned the repo to my computer.
- I clicked new project > existing directory.
- The downside is that RStudio didn't give me the option to make it a Quarto book project.
- In the terminal, following this guidance, I typed `quarto create-project test_quarto_book --type book`.
  - Make sure the terminal is set to the project's directory.
- That added all the built-in book stuff, but in a folder inside the current folder. You can move all of the files you're interested in over if you want. Since I already had the files I had been previously working with (but wasn't able to push to GitHub), I moved them over and worked with them.

11.2 GitHub Pages

Useful websites:

- https://quarto.org/docs/publishing/github-pages.html

That website discusses three methods for publishing the book with GitHub Pages.

1. Render to Docs. The easiest and most straightforward. However, checking in the `_book` directory makes for messy diffs.
2. The publish command. Requires a little set up on the front end, but gives more control.
3. GitHub Action. You might prefer this if you want execution and/or rendering to be automatically triggered from commits. However, it seems like the most complicated option.

I'm going with option 2 for now.

11.3 Netlify

Useful websites:

- https://quarto.org/docs/publishing/netlify.html

Part V

Built-in Chapters

References

Knuth, Donald E. 1984. “Literate Programming.” *Comput. J.* 27 (2): 97–111. https://doi.org/10.1093/comjnl/27.2.97.

A Example appendix

This is just an example appendix.