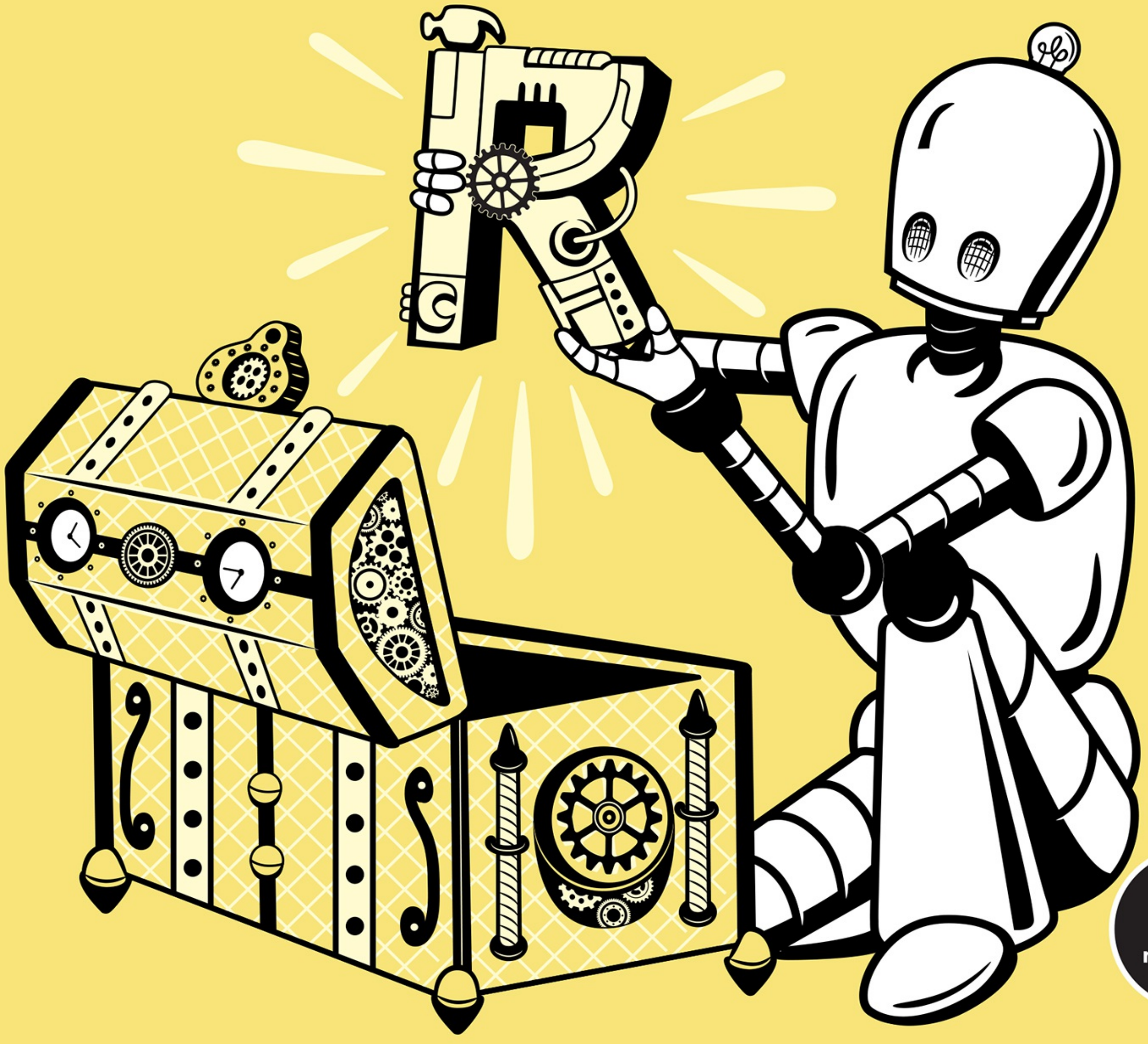
R FOR THE REST OF US

A STATISTICS-FREE INTRODUCTION

DAVID KEYES



no starch press®

CONTENTS IN DETAIL

PRAISE FOR *R FOR THE REST OF US*

"... a fantastic and invaluable resource for anyone working with or aspiring to work with data, regardless of their background. Highly recommended."

—GABRIELA DE QUEIROZ, DIRECTOR OF AI, MICROSOFT

"Long overdue in the R ecosystem."

—OSCAR BARUFFA, *BIG BOOK OF R*

"A great resource for anyone wishing to learn R ..."

—CARA THOMPSON, DATA VISUALIZATION CONSULTANT

"David has captured some of the most compelling ways to use R for data visualization and maps, automation, reporting, and building websites."

—TOM MOCK, PRODUCT MANAGER AT POSIT

"An easy-to-read source of exciting case studies, relevant code snippets, and helpful explanations ..."

—CÉDRIC SCHERER, DATA VISUALIZATION AND INFORMATION GRAPHICS DESIGNER

"Demonstrates how succinctly R can be used to rapidly solve non-statistics problems."

—BOB RUDIS, VP OF DATA SCIENCE, SECURITY RESEARCH & DETECTION ENGINEERING AT GREYNOISE INTELLIGENCE

R FOR THE REST OF US

A Statistics-Free Introduction

by David Keyes

no starch press®

San Francisco

First printing

ISBN-13: 978-1-7185-0332-8 (print)
ISBN-13: 978-1-7185-0333-5 (ebook)

Published by No Starch Press®, Inc.
245 8th Street, San Francisco, CA 94103
phone: +1.415.863.9900
www.nostarch.com; info@nostarch.com

Publisher: William Pollock
Managing Editor: Jill Franklin
Production Manager: Sabrina Plomitallo-González
Production Editor: Sydney Cromwell
Developmental Editor: Frances Saux
Cover Illustrator: Gina Redman
Interior Design: Octopod Studios
Technical Reviewer: Rita Giordano
Copyeditor: Rachel Monaghan
Proofreader: Kim Wimpsett
Indexer: BIM Creatives, LLC

Library of Congress Control Number: 2023051616

[S]

For my wife, Rachel, and my children, Leila and Elias

About the Author

David Keyes is the founder of R for the Rest of Us (*https://rfortherestofus.com*), where he develops courses, conducts corporate trainings, and works with organizations to harness the power of R. While leading a team of consultants, he has overseen the creation of many R-based reports. As a self-taught R user with a qualitative background, he helps people who don't think of themselves as R users learn to use this powerful tool.

About the Technical Reviewer

Rita Giordano is an independent data visualization consultant and LinkedIn instructor based in the UK. She is a physicist and has a PhD in statistics applied to structural biology and has extensive research and data science experience.

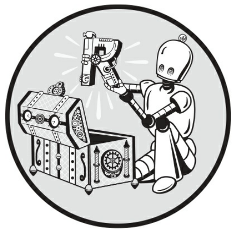
ACKNOWLEDGMENTS

This book is a testament to the many members of the R community who share their knowledge freely and encourage others generously. I call myself self-taught, but really what I am is community-taught. Throughout this book, you will read about several R users from whom I have learned so much; still, many others go unmentioned. To everyone who has worked to develop R, share your knowledge about R, and make R a welcoming place, thank you.

I'd also like to thank the team at R for the Rest of Us. Working directly with talented R users has taught me so much about what is possible with R.

Finally, I'd like to thank people who have provided feedback as I've written this book. Technical reviewer Rita Giordano has helped me make sure everything works and suggested great ideas for improvement. My editor, Frances Saux, has provided fantastic input along the way. To Bill Pollock and the entire team at No Starch: thank you for taking a flyer on me and my strange idea to write a book about nonstatistical uses of a tool created for statistics.

INTRODUCTION



In early 2020, as the world struggled to contain the spread of COVID-19, one country succeeded where others did not: New Zealand. There are many reasons New Zealand was able to tackle COVID-19. One was the R programming language (yes, really).

This humble tool for data analysis helped New Zealand fight COVID-19 by enabling a Ministry of Health team to generate daily reports on cases throughout New Zealand. Based on the information in these reports, officials were able to develop policies that kept the country largely free of COVID-19. The team was small, however, so producing the reports every day with a tool like Excel wouldn't have been feasible. As team leader Chris Knox told me, "Trying to do what we did in a point-and-click environment is not possible."

Instead, a few staff members wrote R code that they could run every day to produce updated reports. These reports did not involve any complicated statistics; they were literally counts of COVID-19 cases. Their value came from everything else that R can do: data analysis and visualization, report creation, and workflow automation.

This book explores the many ways that people use R to communicate and automate tasks. You'll learn how to do the following:

- Make professional-quality data visualizations, maps, and tables
- Replace a clunky multi-tool workflow to create reports with R Markdown

- Use parameterized reporting to generate multiple reports at once
- Produce slideshow presentations and websites using R Markdown
- Automate the process of importing online data from Google Sheets and the US Census Bureau
- Create your own functions to automate tasks you do repeatedly
- Bundle your functions into a package that you can share with others

Best of all, you'll do all of this without performing any statistical analysis more complex than calculating averages

Isn't R Just for Statistical Analysis?

Many people think of R as simply a tool for hardcore statistical analysis, but it can do much more than manipulate numerical values. After all, every R user must illuminate their findings and communicate their results somehow, whether that's via data visualizations, reports, websites, or presentations. Also, the more you use R, the more you'll find yourself wanting to automate tasks you currently do manually.

As a qualitatively trained anthropologist without a quantitative background, I used to feel ashamed about using R for my visualization and communication tasks. But the fact is, R is good at these jobs. The `ggplot2` package is the tool of choice for many top information designers. Users around the world have taken advantage of R's ability to automate reporting to make their work more efficient. Rather than simply replacing other tools, R can perform tasks that you're probably already doing, like generating reports and tables, *better* than your existing workflow.

Who This Book Is For

No matter your background, using R can transform your work. This book is for you if you're either a current R user keen to explore its uses for visualization and communication or a non-R user wondering if R is right for you. I've written *R for the Rest of Us* so that it should make sense whether or not you've ever written a line of R code. But even if you've written entire R programs, the book should help you learn plenty of new techniques to up your game.

R is a great tool for anyone who works with data. Maybe you're a researcher looking for a new way to share your results. Perhaps you're a journalist looking to analyze public data more efficiently. Or maybe you're a data analyst tired of working in expensive, proprietary tools. If you have to work with data, you will get value from R.

About This Book

Each chapter focuses on one use of the R language and includes examples of real R projects that employ the techniques covered. I'll dive into the project code, breaking the programs down to help you understand how they work, and suggest ways of going beyond the example. The book has three parts, outlined here.

In Part I, you'll learn how to use R to visualize data.

Chapter 1: An R Programming Crash Course Introduces the RStudio programming environment and the foundational R syntax you'll need to understand the rest of the book.

Chapter 2: Principles of Data Visualization Breaks down a visualization created for *Scientific American* on drought conditions in the United States. In doing so, this chapter introduces the `ggplot2` package for data visualization and addresses important principles that can help you make high-quality graphics.

Chapter 3: Custom Data Visualization Themes Describes how journalists at the BBC made a custom theme for the `ggplot2` data visualization package. As the chapter walks you through the package they created, you'll learn how to make your own theme.

Chapter 4: Maps and Geospatial Data Explores the process of making maps in R using simple features data. You'll learn how to write map-making code, find geospatial data, choose appropriate projections, and apply data visualization principles to make your map appealing.

Chapter 5: Designing Effective Tables Shows you how to use the `gt` package to make high-quality tables in R. With guidance from R table connoisseur Tom Mock, you'll learn the design principles to present your table data effectively.

Part II focuses on using R Markdown to communicate efficiently. You'll learn how to incorporate visualizations like the ones discussed in Part I into reports, slideshow presentations, and static websites generated entirely using R code.

Chapter 6: R Markdown Reports Introduces R Markdown, a tool that allows you to generate a professional report in R. This chapter covers the structure of an R Markdown document, shows you how to use inline code to automatically update your report's text when data values change, and discusses the tool's many export options.

Chapter 7: Parameterized Reporting Covers one of the advantages of using R Markdown: the ability to produce multiple reports at the same time using a technique called *parameterized reporting*. You'll see how staff members at the Urban Institute used R to generate fiscal briefs for all 50 US states. In the process, you'll learn how parameterized reporting works and how you can use it.

Chapter 8: Slideshow Presentations Explains how to use R Markdown to make slides with the `xaringan` package. You'll learn how to make your own presentations, adjust your content to fit on a slide, and add effects to your slideshow.

Chapter 9: Websites Shows you how to create your own website with R Markdown and the `distill` package. By examining a website about COVID-19 rates in Westchester County, New York, you'll see how to create pages on your site, add interactivity through R packages, and deploy your website in multiple ways.

Chapter 10: Quarto Explains how to use Quarto, the next-generation version of R Markdown. You'll learn how to use Quarto for all of the projects you previously used R Markdown for (reports, parameterized reporting, slideshow presentations, and websites).

Part III focuses on ways you can use R to automate your work and share it with others.

Chapter 11: Automatically Accessing Online Data Explores two R packages that let you automatically import data from the internet: `googlesheets4` for working with Google Sheets and `tidycensus` for working with US Census Bureau data. You'll learn how the packages work and how to use them to automate the process of accessing data.

Chapter 12: Creating Functions and Packages Shows you how to create your own functions and packages and share them with others, which is one of R's major benefits. Bundling your custom functions into a package can enable other R users to streamline their work, as you'll read about with the packages that a group of R developers built for researchers working at the Moffitt Cancer Center.

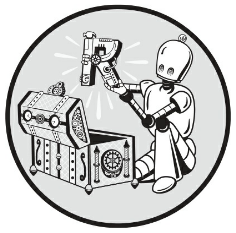
By the end of this book, you should be able to use R for a wide range of nonstatistical tasks. You'll know how to effectively visualize data and communicate your findings using maps and tables. You'll be able to integrate your results into reports using R Markdown, as well as efficiently generate slideshow presentations and websites. And you'll understand how to automate many tedious tasks using packages others have built or ones you develop yourself. Let's dive in!

PART I

VISUALIZATIONS

1

AN R PROGRAMMING CRASH COURSE



R has a well-earned reputation for being hard to learn, especially for those who come to it without prior programming experience. This chapter is designed to help anyone who has never used R before. You’ll set up an R programming environment with RStudio and learn how to use functions, objects, packages, and projects to work with data. You’ll also be introduced to the `tidyverse` package, which contains the core data analysis and manipulation functions used in this book.

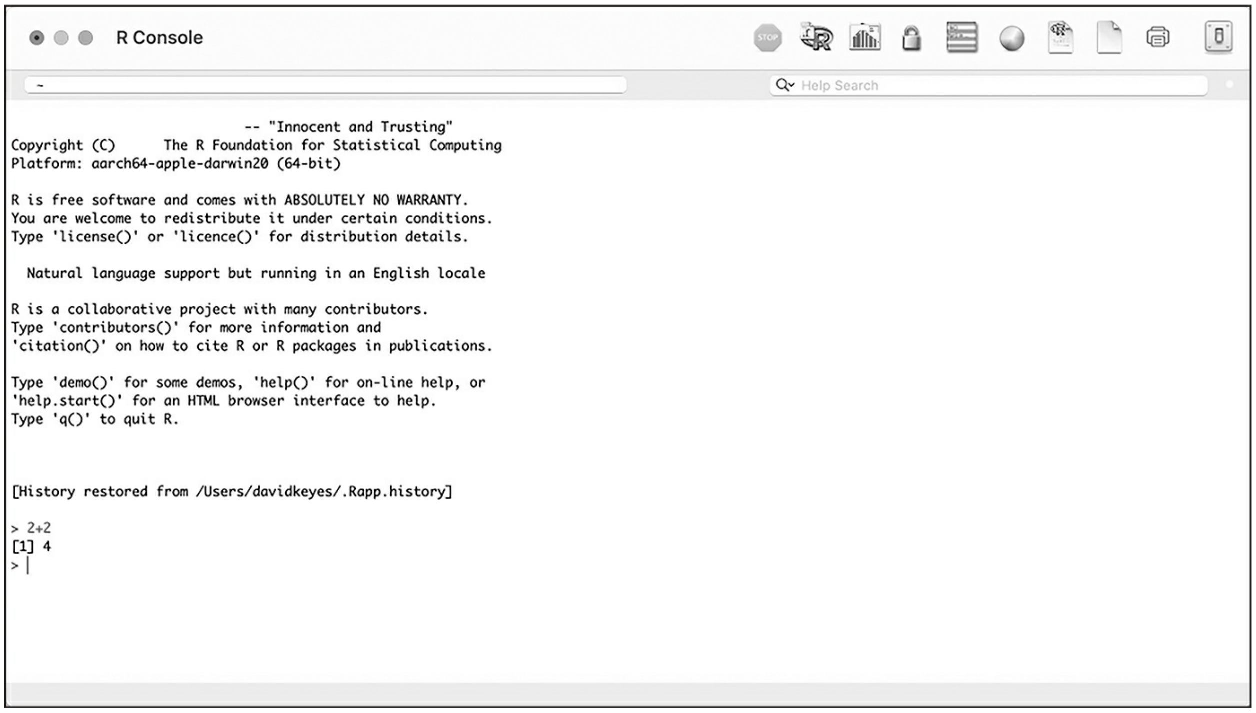
This chapter won’t provide a complete introduction to R programming; rather, it will focus on the knowledge you need to follow along with the rest of the book. If you have prior experience with R, feel free to skip ahead to Chapter 2.

Setting Up

You’ll need two pieces of software to use R effectively. The first is R itself, which provides the underlying computational tools that make the language work. The second is an *integrated development environment (IDE)* like RStudio. This coding platform simplifies working with R. The best way to understand the relationship between R and RStudio is with this analogy from Chester Ismay and Albert Kim's book *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse*: R is the engine that powers your data, and RStudio is like the dashboard that provides a user-friendly interface.

Installing R and RStudio

To download R, go to *https://cloud.r-project.org* and choose the link for your operating system. Once you've installed it, open the file. This should open an interface, like the one shown in Figure 1-1, that lets you work with R on your operating system's command line. For example, enter **`2 + 2`**, and you should see `4`.



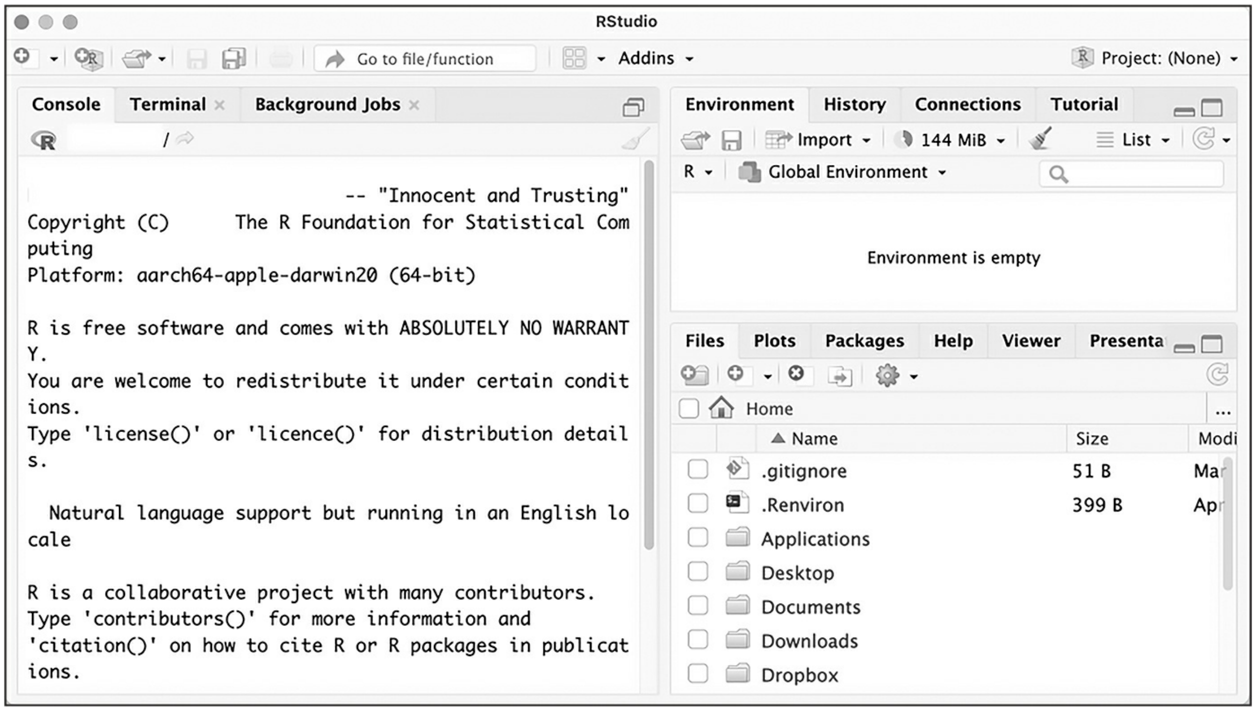
```
                              -- "Innocent and Trusting"
Copyright (C)      The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.


[History restored from /Users/davidkeyes/.Rapp.history]

> 2+2
[1] 4
> |
```

Figure 1-1: The R console

A few brave souls work with R using only this command line, but most opt to use RStudio, which provides a way to see your files, the output of your code, and more. You can download RStudio at *https://posit.co/download/rstudio-desktop/*. Install RStudio as you would any other app and open it.

Exploring the RStudio Interface

The first time you open RStudio, you should see the three panes shown in Figure 1-2.



```
                              -- "Innocent and Trusting"
Copyright (C)      The R Foundation for Statistical Com
puting
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANT
Y.
You are welcome to redistribute it under certain condit
ions.
Type 'license()' or 'licence()' for distribution detail
s.

  Natural language support but running in an English lo
cale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publicat
ions.
```

Figure 1-2: The RStudio editor

The left pane should look familiar. It’s similar to the screen you saw when working in R on the command line. This is known as the *console*. You’ll use it to enter code and see the results. This pane has several tabs, such as Terminal and Background Jobs, for more advanced uses. For now, you’ll stick to the default tab.

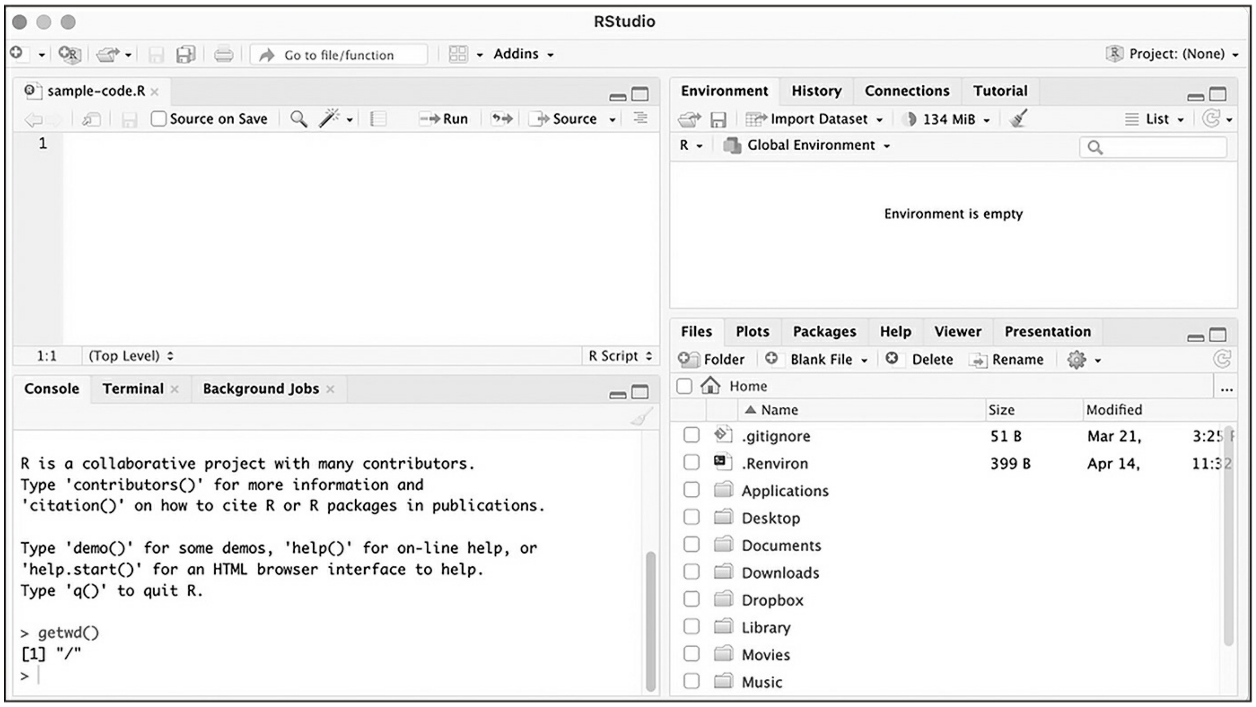
At the bottom right, the *files pane* shows all of the files on your computer. You can click any file to open it within RStudio. Finally, at the top right is the *environment pane*, which shows the objects that are available to you when working in RStudio. Objects are discussed in “Saving Data as Objects” on page 11.

There is one more pane that you’ll typically use when working in RStudio, but to see it, first you need to create an R script file.

R Script Files

If you write all of your code in the console, you won’t have any record of it. Say you sit down today and import your data, analyze it, and then make some graphs. If you run these operations in the console, you’ll have to re-create that code from scratch tomorrow. But if you write your code in files instead, you can run it multiple times.

R script files, which use the *.R* extension, save your code so you can run it later. To create an R script file, go to **File ▸ New File ▸ R Script**, and the *script file pane* should appear in the top left of RStudio, as shown in Figure 1-3. Save this file in your *Documents* folder as *sample-code.R*.



```
R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> getwd()
[1] "/"
>
```

Figure 1-3: The script file pane (top left)

Now you can enter R code into the new pane to add it to your script file. For example, try entering `2 + 2` in the script file pane to perform a simple addition operation.

To run a script file, click **Run** or use the keyboard shortcut COMMAND-ENTER on macOS or CTRL-ENTER on Windows. The result (4, in this case) should show up in the console pane.

You now have a working programming environment. Next you’ll use it to write some simple R code.

Basic R Syntax

If you’re trying to learn R, you probably want to perform more complex operations than `2 + 2`, but understanding the fundamentals will prepare you to do more serious data analysis tasks later in this chapter. Let’s cover some of these basics.

Arithmetic Operators

Besides `+`, R supports the common arithmetic operators `-` for subtraction, `*` for multiplication, and `/` for division. Try entering the following in the console:

```
> 2 - 1
1
> 3 * 3
9
> 16 / 4
4
```

As you can see, R returns the result of each calculation you enter. You don’t have to add the spaces around operators as shown here, but doing so makes your code much more readable.

You can also use parentheses to perform multiple operations at once and see their result. The parentheses specify the order in which R will evaluate the expression. Try running the following:

```
> 2 * (2 + 1)
6
```

This code first evaluates the expression within the parentheses, `2 + 1`, before multiplying the result by `2` in order to get `6`.

R also has more advanced arithmetic operators, such as `**` to calculate exponents:

```
> 2 ** 3
8
```

This is equivalent to 2^3, which returns `8`.

To get the remainder of a division operation, you can use the `%%` operator:

```
> 10 %% 3
1
```

Dividing `10` by `3` produces a remainder of `1`, the value R returns.

You won't need to use these advanced arithmetic operators for the activities in this book, but they're good to know nonetheless.

Comparison Operators

R also uses *comparison operators*, which let you test how one value compares to another. R will return either `TRUE` or `FALSE`. For example, enter **`2 > 1`** in the console:

```
> 2 > 1
TRUE
```

R should return `TRUE`, because `2` is greater than `1`.

Other common comparison operators include less than (<), greater than or equal to (>=), less than or equal to (<=), equal to (==), and not equal to (`!=`). Here are some examples:

```
> 498 == 498
TRUE
> 2 != 2
FALSE
```

When you enter `498 == 498` in the console, R should return `TRUE` because the two values are equal. If you run `2 != 2` in the console, R should return `FALSE` because `2` does not *not* equal `2`.

You'll rarely use comparison operators to directly test how one value compares to another; instead, you'll use them to perform tasks like keeping only data where a value is greater than a certain threshold. You'll see comparison operators used in this way in "tidyverse Functions" on page 15.

Functions

You can perform even more useful operations by making use of R's many *functions*, predefined sections of code that let you efficiently do specific things. Functions have a name and a set of parentheses containing *arguments*, which are values that affect the function's behavior.

Consider the `print()` function, which displays information:

```
> print(x = 1.1)
1.1
```

The name of the `print()` function is `print`. Within the function's parentheses, you specify the argument name—`x`, in this case—followed by the equal sign (=) and a value for the function to display. This code will print the number `1.1`.

To separate multiple arguments, you use commas. For example, you can use the `print()` function's `digits` argument to indicate how many digits of a number to display:

```
> print(x = 1.1, digits = 1)
1
```

This code will display only one digit (in other words, a whole number).

Using these two arguments allows you to do something specific (display results) while also giving you the flexibility to change the function's behavior.

NOTE

For a list of all functions built into R, see https://stat.ethz.ch/R-manual/R-devel/library/base/html/00Index.xhtml.

A common R pattern is using a function within a function. For example, if you wanted to calculate the *mean*, or average, of the values 10, 20, and 30, you could use the `mean()` function to operate on the result of the `c()` function like so:

```
> mean(x = c(10, 20, 30))
```

The `c()` function combines multiple values into one, which is necessary because the `mean()` function accepts only one argument. This is why the code has two matching sets of open and close parentheses: one for `mean()` and a nested one for `c()`.

The value after the equal sign in this example, `c(10, 20, 30)`, tells R to use the values 10, 20, and 30 to calculate the mean. Running this code in the console returns the value `20`.

The functions `median()` and `mode()` work with `c()` in the same way. To learn how to use a function and what arguments it accepts, enter **`?`** followed by the function's name in the console to see the function's help file.

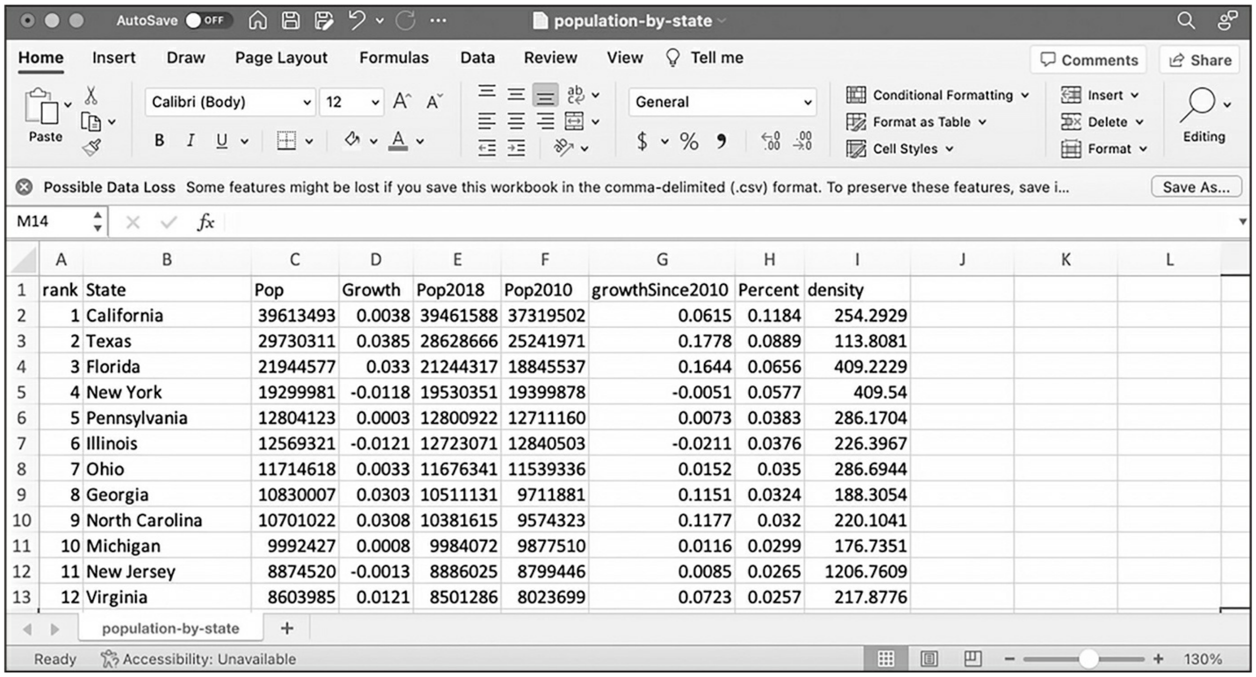
Next, let's look at how to import data for your R programs to work with.

Working with Data

R lets you do all of the same data manipulation tasks you might perform in a tool like Excel, such as calculating averages or totals. Conceptually, however, working with data in R is very different from working with Excel, where your data and analysis code live in the same place: a spreadsheet. While the data you work with in R might look similar to the data you work with in Excel, it typically comes from some external file, so you have to run code to import it.

Importing Data

You'll import data from a *comma-separated values (CSV)* file, a text file that holds a series of related values separated by commas. You can open CSV files using most spreadsheet applications, which use columns rather than commas as separators. For example, Figure 1-4 shows the *population-by-state.csv* file in Excel.



Possible Data Loss Some features might be lost if you save this workbook in the comma-delimited (.csv) format. To preserve these features, save i...

rank	State	Pop	Growth	Pop2018	Pop2010	growthSince2010	Percent	density
1	California	39613493	0.0038	39461588	37319502	0.0615	0.1184	254.2929
2	Texas	29730311	0.0385	28628666	25241971	0.1778	0.0889	113.8081
3	Florida	21944577	0.033	21244317	18845537	0.1644	0.0656	409.2229
4	New York	19299981	-0.0118	19530351	19399878	-0.0051	0.0577	409.54
5	Pennsylvania	12804123	0.0003	12800922	12711160	0.0073	0.0383	286.1704
6	Illinois	12569321	-0.0121	12723071	12840503	-0.0211	0.0376	226.3967
7	Ohio	11714618	0.0033	11676341	11539336	0.0152	0.035	286.6944
8	Georgia	10830007	0.0303	10511131	9711881	0.1151	0.0324	188.3054
9	North Carolina	10701022	0.0308	10381615	9574323	0.1177	0.032	220.1041
10	Michigan	9992427	0.0008	9984072	9877510	0.0116	0.0299	176.7351
11	New Jersey	8874520	-0.0013	8886025	8799446	0.0085	0.0265	1206.7609
12	Virginia	8603985	0.0121	8501286	8023699	0.0723	0.0257	217.8776

Figure 1-4: The population-by-state.csv *file in Excel*

To work with this file in R, download it from *https://data.rfortherestofus.com/population-by-state.csv*. Save it to a location on your computer, such as your *Documents* folder.

Next, to import the file into R, add a line like the following to the *sample-code.R* file you created earlier in this chapter, replacing my filepath with the path to the file's location on your system:

```
read.csv(file = "/Users/davidkeyes/Documents/population-by-st
ate.csv")
```

The `file` argument in the `read.csv()` function specifies the path to the file to open.

The `read.csv()` function can accept additional optional arguments, separated by commas. For example, the following line uses the `skip` argument in addition to `file` to import the same file but skip the first row:

```
read.csv(file = "/Users/davidkeyes/Documents/population-by-st
ate.csv", skip = 1)
```

To learn about additional arguments for this function, enter **`?read.csv()`** in the console to see its help file.

At this point, you can run the code to import your data (without the `skip` argument). Highlight the line you want to run in the script file pane in RStudio and click **Run**. You should see the following output in the console pane:

```
#>    rank                State      Pop  Growth  Pop2018
#> 1     1           California 39613493  0.0038 39461588
#> 2     2                Texas 29730311  0.0385 28628666
#> 3     3              Florida 21944577  0.0330 21244317
#> 4     4             New York 19299981 -0.0118 19530351
#> 5     5         Pennsylvania 12804123  0.0003 12800922
#> 6     6             Illinois 12569321 -0.0121 12723071
#> 7     7                 Ohio 11714618  0.0033 11676341
#> 8     8              Georgia 10830007  0.0303 10511131
#> 9     9       North Carolina 10701022  0.0308 10381615
#> 10   10             Michigan  9992427  0.0008  9984072
--snip--
```

This is R's way of confirming that it imported the CSV file and understands the data within it. Four variables show each state's rank (in terms of population size), name, current population, population growth between the `Pop` and `Pop2018` variables (expressed as a percentage), and 2018 population. Several other variables are hidden in the output, but you'll see them if you import this CSV file yourself.

You might think you're ready to work with your data now, but all you've really done at this point is display the result of running the code that imports the data. To actually use the data, you need to save it to an object.

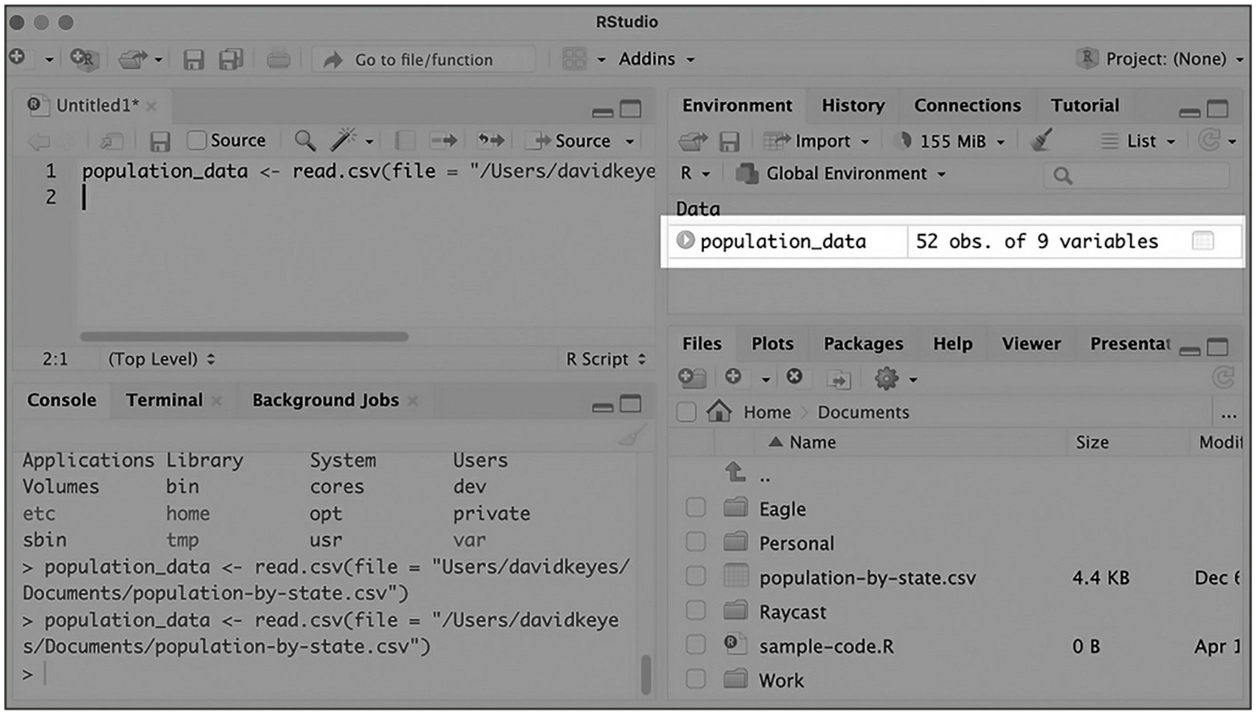
Saving Data as Objects

To save your data for reuse, you need to create an object. For the purposes of this discussion, an *object* is a data structure that is stored for later use. To create an object, update your data-importing syntax so it looks like this:

```
population_data <- read.csv(file = "/Users/davidkeyes/Documen
ts/population-by-state.csv")
```

Now this line of code contains the `<-` *assignment operator*, which takes what follows it and assigns it to the item on the left. To the left of the assignment operator is the `population_data` object. Put together, the whole line imports the CSV file and assigns it to an object called `population_data`.

When you run this code, you should see `population_data` in your environment pane, as shown in Figure 1-5.



```
population_data <- read.csv(file = "/Users/davidkeye
```

```
Applications Library      System      Users
Volumes      bin          cores       dev
etc          home         opt         private
sbin         tmp          usr         var
> population_data <- read.csv(file = "Users/davidkeyes/
Documents/population-by-state.csv")
> population_data <- read.csv(file = "/Users/davidkeye
s/Documents/population-by-state.csv")
>
```

```
population_data    52 obs. of 9 variables
```

Name	Size	Modi
..		
Eagle		
Personal		
population-by-state.csv	4.4 KB	Dec 6
Raycast		
sample-code.R	0 B	Apr 1
Work		

Figure 1-5: The population_data object in the environment pane

This message confirms that your data import worked and that the `population_data` object is ready for future use. Now, instead of having to rerun the code to import the data, you can simply enter `population_data` in an R script file or in the console to output the data.

Data imported to an object in this way is known as a *data frame*. You can see that the `population_data` data frame has 52 observations and 9 variables. *Variables* are the data frame’s columns, each of which represents some value (for example, the population of each state). As you’ll see throughout the book, you can add new variables or modify existing ones using R code. The 52 observations come from the 50 states, as well as the District of Columbia and Puerto Rico.

Installing Packages

The read.csv() function you've been using, as well as the mean() and c() functions you saw earlier, comes from *base R*, the set of built-in R functions. To use base R functions, you simply enter their names. However, one of the benefits of R being an open source language is that anyone can create their own code and share it with others. R users around the world make R *packages*, which provide custom functions to accomplish specific goals.

The best analogy for understanding packages also comes from the book *Statistical Inference via Data Science*. The functionality in base R is like the features built into a smartphone. A smartphone can do a lot on its own, but you usually want to install additional apps for specific tasks. Packages are like apps, giving you functionality beyond what's built into base R. In Chapter 12, you'll create your own R package.

You can install packages using the install.packages() function. You'll be working with the tidyverse package, which provides a range of functions for data import, cleaning, analysis, visualization, and more. To install it, enter **install.packages("tidyverse")**. Typically, you'll enter package installation code in the console rather than in a script file because you need to install a package only once on your computer to access its code in the future.

To confirm that the tidyverse package has been installed correctly, click the **Packages** tab on the bottom-right pane in RStudio. Search for tidyverse, and you should see it pop up.

Now that you've installed the tidyverse, you'll put it to use. Although you need to *install* packages only once per computer, you need to *load* them each time you restart RStudio. Return to the *sample-code.R* file and reimport your data using a function from the tidyverse package (your filepath will look slightly different):

```
library(tidyverse)

population_data_2 <- read_csv(file = "/Users/davidkeyes/Docum
ents/population-by-state.csv")
```

At the top of the script, the line library(tidyverse) loads the `tidyverse` package. Then, the package's `read_csv()` function imports the data. Note the underscore (_) in place of the period (`.`) in the function's name; this differs from the base R function you used earlier. Using `read_csv()` to import CSV files achieves the same goal of creating an object, however—in this case, one called `population_data_2`. Enter **`population_data_2`** in the console, and you should see this output:

```
#> # A tibble: 52 × 9
#>     rank State               Pop  Growth  Pop2018  Pop2010
#>    <dbl> <chr>             <dbl>   <dbl>    <dbl>    <dbl>
#>  1     1 California     39613493  0.0038 39461588 37319502
#>  2     2 Texas          29730311  0.0385 28628666 25241971
#>  3     3 Florida        21944577  0.0330 21244317 18845537
#>  4     4 New York       19299981 -0.0118 19530351 19399878
#>  5     5 Pennsylvania   12804123  0.0003 12800922 12711160
#>  6     6 Illinois       12569321 -0.0121 12723071 12840503
#>  7     7 Ohio           11714618  0.0033 11676341 11539336
#>  8     8 Georgia        10830007  0.0303 10511131  9711881
#>  9     9 North Carolina 10701022  0.0308 10381615  9574323
#> 10    10 Michigan        9992427  0.0008  9984072  9877510
#> # 42 more rows
#> # 3 more variables: growthSince2010 <dbl>, Percent <dbl>,
#> #   density <dbl>
```

This data looks slightly different from the data you generated using the `read.csv()` function. For example, R shows only the first 10 rows. This variation occurs because `read_csv()` imports the data not as a data frame but as a data type called a *tibble*. Both data frames and tibbles are used to describe *rectangular* data like what you would see in a spreadsheet. There are some minor differences between data frames and tibbles, the most important of which is that tibbles print only the first 10 rows by default, while data frames print all rows. For the purposes of this book, the two terms are used interchangeably.

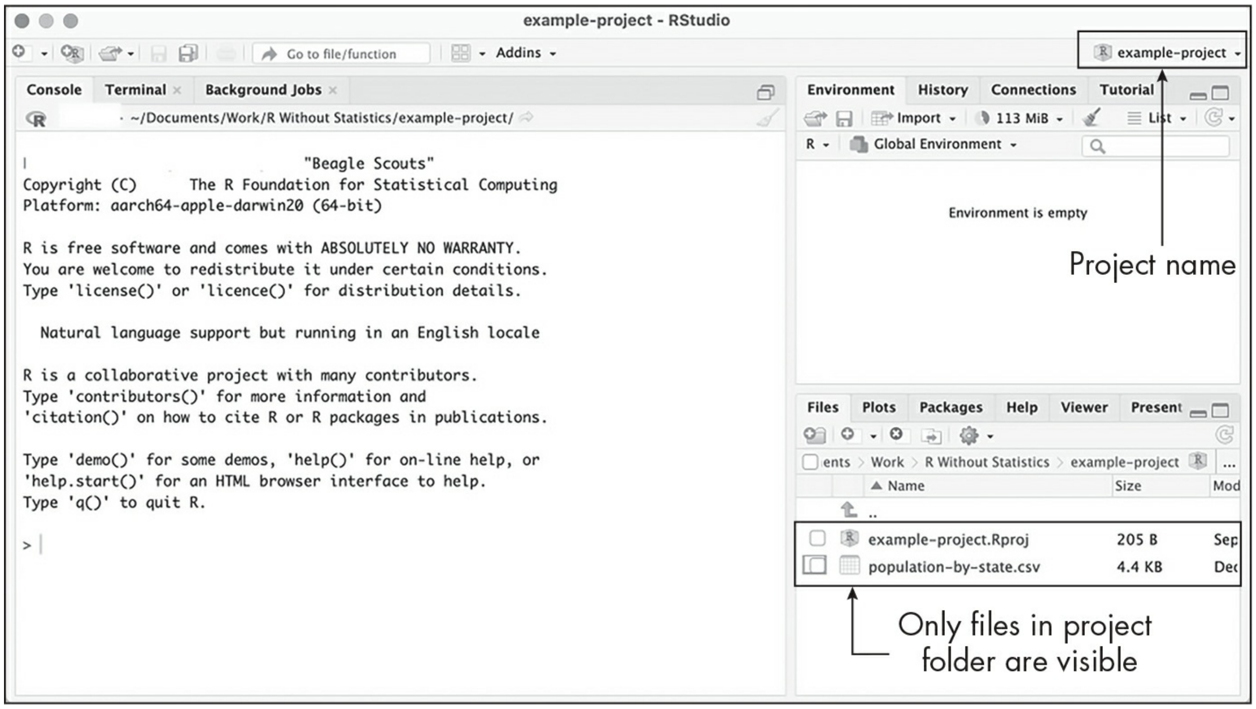
RStudio Projects

So far, you've imported a CSV file from your *Documents* folder. But because others won't have this exact location on their computer, your code won't work if they try to run it. One solution to this problem is an RStudio project.

By working in a project, you can use *relative paths* to your files instead of having to write the entire filepath when calling a function to import data. Then, if you place the CSV file in your project, anyone can open it by using the file's name, as in `read_csv(file = "population-by-state.csv")`. This makes the path easier to write and enables others to use your code.

To create a new RStudio project, go to **File ▸ New Project**. Select either **New Directory** or **Existing Directory** and choose where to put your project. If you choose New Directory, you'll need to specify that you want to create a new project. Next, choose a name for the new directory and where it should live. (Leave the checkboxes that ask about creating a Git repository and using `renv` unchecked; they're for more advanced purposes.)

Once you've created this project, you should see two major differences in RStudio's appearance. First, the files pane no longer shows every file on your computer. Instead, it shows only files in the *example-project* directory. Right now, that's just the *example-project.Rproj* file, which indicates that the folder contains a project. Second, at the top right of RStudio, you can see the name *example-project*. This label previously read `Project: (None)`. If you want to make sure you're working in a project, check for its name here. Figure 1-6 shows these changes.



```
|                          "Beagle Scouts"
Copyright (C)      The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> |
```

Name	Size	Mod
example-project.Rproj	205 B	Sep
population-by-state.csv	4.4 KB	Dec

Figure 1-6: RStudio with an active project

Now that you've created a project, copy the *population-by-state.csv* file into the *example-project* directory. Once you've done so, you should see it in the RStudio files pane.

With this CSV file in your project, you can now import it more easily. As before, start by loading the `tidyverse` package. Then, remove the reference to the *Documents* folder and import your data by simply using the name of the file:

```
library(tidyverse)

population_data_2 <- read_csv(file = "population-by-state.csv
")
```

The reason you can import the *population-by-state.csv* file this way is that the RStudio project sets the working directory to be the root of your project. With the working directory set like this, all references to files are relative to the *.Rproj* file at the root of the project. Now anyone can run this code because it imports the data from a location that is guaranteed to exist on their computer.

Data Analysis with the tidyverse

Now that you've imported the population data, you're ready to do a bit of analysis on it. Although I've been referring to the `tidyverse` as a single package, it's actually a collection of packages. We'll explore several of its functions throughout this book, but this section introduces you to its basic workflow.

tidyverse Functions

Because you've loaded the `tidyverse` package, you can now access its functions. For example, the package's `summarize()` function takes a data frame or tibble and calculates some piece of information for one or more of the variables in that dataset. The following code uses `summarize()` to calculate the mean population of all states:

```
summarize(.data = population_data_2, mean_population = mean(P
op))
```

First, the code passes `population_data_2` to the `summarize()` function's `.data` argument to tell R to use that data frame to perform the calculation. Next, it creates a new variable called `mean_population` and assigns it to the output of the `mean()` function introduced earlier. The `mean()` function runs on `Pop`, one of the variables in the `population_data_2` data frame.

You might be wondering why you don't need to use the `c()` function within `mean()`, as shown earlier in this chapter. The reason is that you're passing the function only one argument here: `Pop`, which contains the set of population data for which you're calculating the mean. In this case, there's no need to use `c()` to combine multiple values into one.

Running this code should return a tibble with a single variable (`mean_population`), as shown here:

```
#> # A tibble: 1 × 1
#>   mean_population
#>             <dbl>
#> 1         6433422
```

The variable is of type double (`dbl`), which is used to hold general numeric data. Other common data types are integer (for whole numbers, such as `4`, `82`, and `915`), character (for text values), and logical (for the `TRUE`/`FALSE` values returned from comparison operations). The `mean_population` variable has a value of `6433422`, the mean population of all states.

Notice also that the `summarize()` function creates a totally new tibble from the original `population_data_2` data frame. This is why the variables from `population_data_2` are no longer present in the output.

This is a basic example of data analysis, but you can do a lot more with the `tidyverse`.

The tidyverse Pipe

One advantage of working with the `tidyverse` is that it uses the *pipe* for multistep operations. The `tidyverse` pipe, which is written as `%>%`, allows you to break steps into multiple lines. For example, you could rewrite your code using the pipe like so:

```
population_data_2 %>%
  summarize(mean_population = mean(Pop))
```

This code says, “Start with the `population_data_2` data frame, then run the `summarize()` function on it, creating a variable called `mean_population` by calculating the mean of the `Pop` variable.”

Notice that the line following the pipe is indented. To make the code easier to read, RStudio automatically adds two spaces to the start of lines that follow pipes.

The pipe becomes even more useful when you use multiple steps in your data analysis. Say, for example, you want to calculate the mean population of the five largest states. The following code adds a line that uses the `filter()` function, also from the `tidyverse` package, to include only states where the `rank` variable is less than or equal to (<=) `5`. Then, it uses `summarize()` to calculate the mean of those states:

```
population_data_2 %>%
  filter(rank <= 5) %>%
  summarize(mean_population = mean(Pop))
```

Running this code returns the mean population of the five largest states:

```
#> # A tibble: 1 × 1
#>   mean_population
#>             <dbl>
#> 1        24678497
```

Using the pipe to combine functions lets you refine your data in multiple ways while keeping it readable and easy to understand. Indentation can also make your code more readable. You’ve seen only a few functions for analysis at this point, but the `tidyverse` has many more functions that enable you to do nearly anything you could hope to do with your data. Because of how useful the `tidyverse` is, it will appear in every single piece of R code you write in this book.

NOTE

R for Data Science, *2nd edition, by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund is the bible of `tidyverse` programming and worth reading for more details on how the package's many functions work.*

Comments

In addition to code, R script files often contain *comments*—lines that begin with hash marks (#) and aren't treated as runnable code but instead as notes for anyone reading the script. For example, you could add a comment to the code from the previous section, like so:

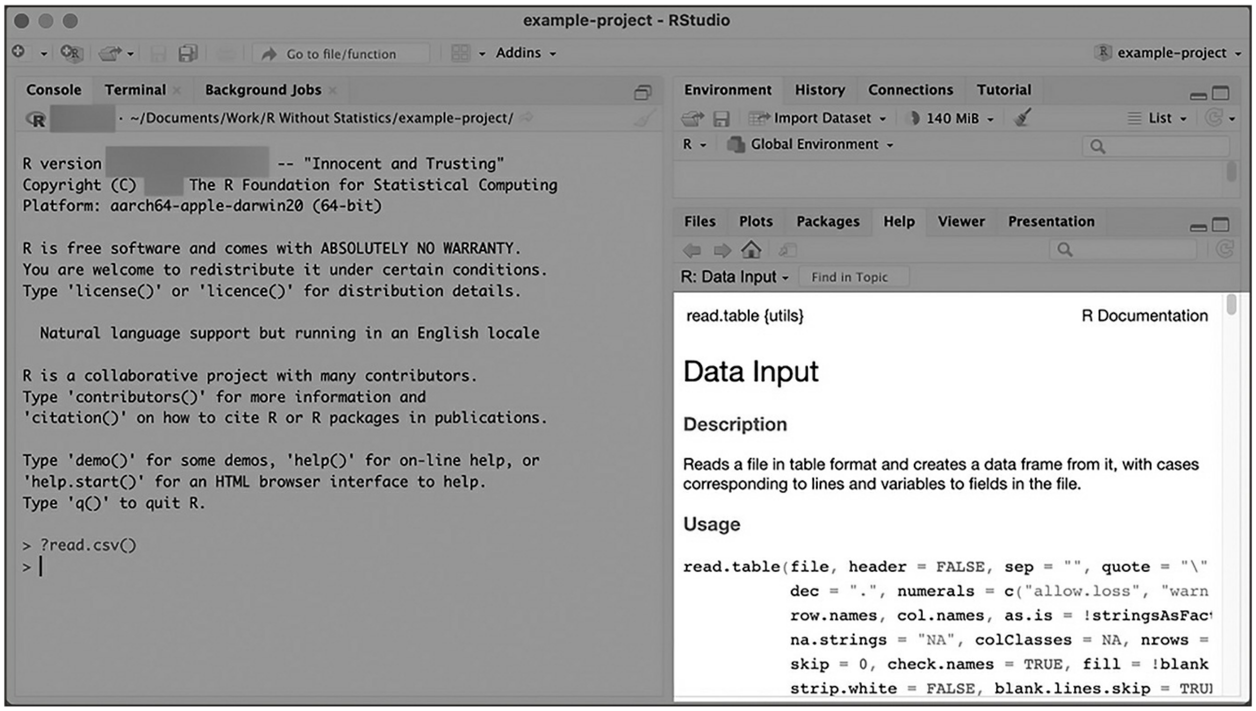
```
# Calculate the mean population of the five largest states

population_data_2 %>%
  filter(rank <= 5) %>%
  summarize(mean_population = mean(Pop))
```

This comment will help others understand what is happening in the code, and it can also serve as a useful reminder for you if you haven't worked on the code in a while. R knows to ignore any lines that begin with the hash mark instead of trying to run them.

How to Get Help

Now that you've learned the basics of how R works, you're probably ready to dive in and write some code. When you do, though, you're going to encounter errors. Being able to get help when you run into issues is a key part of learning to use R successfully. There are two main strategies you can use to get unstuck.

The first is to read the documentation for the functions you use. Remember, to access the documentation for any function, simply enter **`?`** and then the name of the function in the console. In the bottom-right pane in Figure 1-7, for example, you can see the result of running `?read.csv`.



[redacted] · ~/Documents/Work/R Without Statistics/example-project/

```
R version [redacted] -- "Innocent and Trusting"
Copyright (C) [redacted] The R Foundation for Statistical Computing
Platform: aarch64-apple-darwin20 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

> ?read.csv()
> |
```

read.table {utils} R Documentation

Data Input

Description

Reads a file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

Usage

```
read.table(file, header = FALSE, sep = "", quote = "\"
           dec = ".", numerals = c("allow.loss", "warn
           row.names, col.names, as.is = !stringsAsFac
           na.strings = "NA", colClasses = NA, nrows =
           skip = 0, check.names = TRUE, fill = !blank
           strip.white = FALSE, blank.lines.skip = TRUI
```

Figure 1-7: The documentation for the `read.csv()` function

Help files can be a bit hard to decipher, but essentially they describe what package the function comes from, what the function does, what arguments it accepts, and some examples of how to use it.

NOTE

For additional guidance on reading documentation, I recommend the appendix of Kieran Healy's book Data Visualization: A Practical Introduction. *A free online version is available at* https://socviz.co/appendix.xhtml.

The second approach is to read the documentation websites associated with many R packages. These can be easier to read than RStudio's help files. In addition, they often contain longer articles, known as *vignettes*, that provide an overview of how a given package works. Reading these can help you understand how to combine individual functions in the context of a larger project. Every package discussed in this book has a good documentation website.

Summary

In this chapter, you learned the basics of R programming. You saw how to download and set up R and RStudio, what the various RStudio panes are for, and how R script files work. You also learned how to import CSV files and explore them in R, how to save data as objects, and how to install packages to access additional functions. Then, to make the files used in your code more accessible, you created an RStudio project. Finally, you experimented with `tidyverse` functions and the `tidyverse` pipe, and you learned how to get help when those functions don't work as expected.

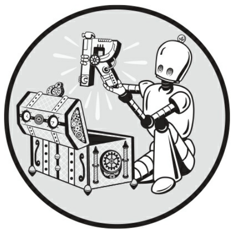
Now that you understand the basics, you're ready to start using R to work with your data. See you in Chapter 2!

Additional Resources

- Kieran Healy, *Data Visualization: A Practical Introduction* (Princeton, NJ: Princeton University Press, 2018), *https://socviz.co*.
- Chester Ismay and Albert Y. Kim, *Statistical Inference via Data Science: A ModernDive into R and the Tidyverse* (Boca Raton, FL: CRC Press, 2020), *https://moderndive.com*.
- David Keyes, "Getting Started with R," online course, accessed November 10, 2023, *https://rfortherestofus.com/courses/getting-started*.
- Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund, *R for Data Science*, 2nd ed. (Sebastopol, CA: O'Reilly Media, 2023).

2

PRINCIPLES OF DATA VISUALIZATION



In the spring of 2021, nearly all of the American West was in a drought. Officials in Southern California declared a water emergency in April, citing unprecedented conditions. This probably didn't come as news to residents of California and other western states. Drought conditions like those in the West in 2021 are becoming increasingly common, yet communicating the extent of the problem remains difficult. How can this data be presented in a way that is both accurate and compelling enough to get people to take notice?

Data visualization designers Cédric Scherer and Georgios Karamanis took on this challenge in the fall of 2021 to create a graph of US drought conditions over the last two decades for the magazine *Scientific American*. They turned to the `ggplot2` package to transform dry data (pardon the pun) from the National Drought Center into a visually arresting and impactful visualization.

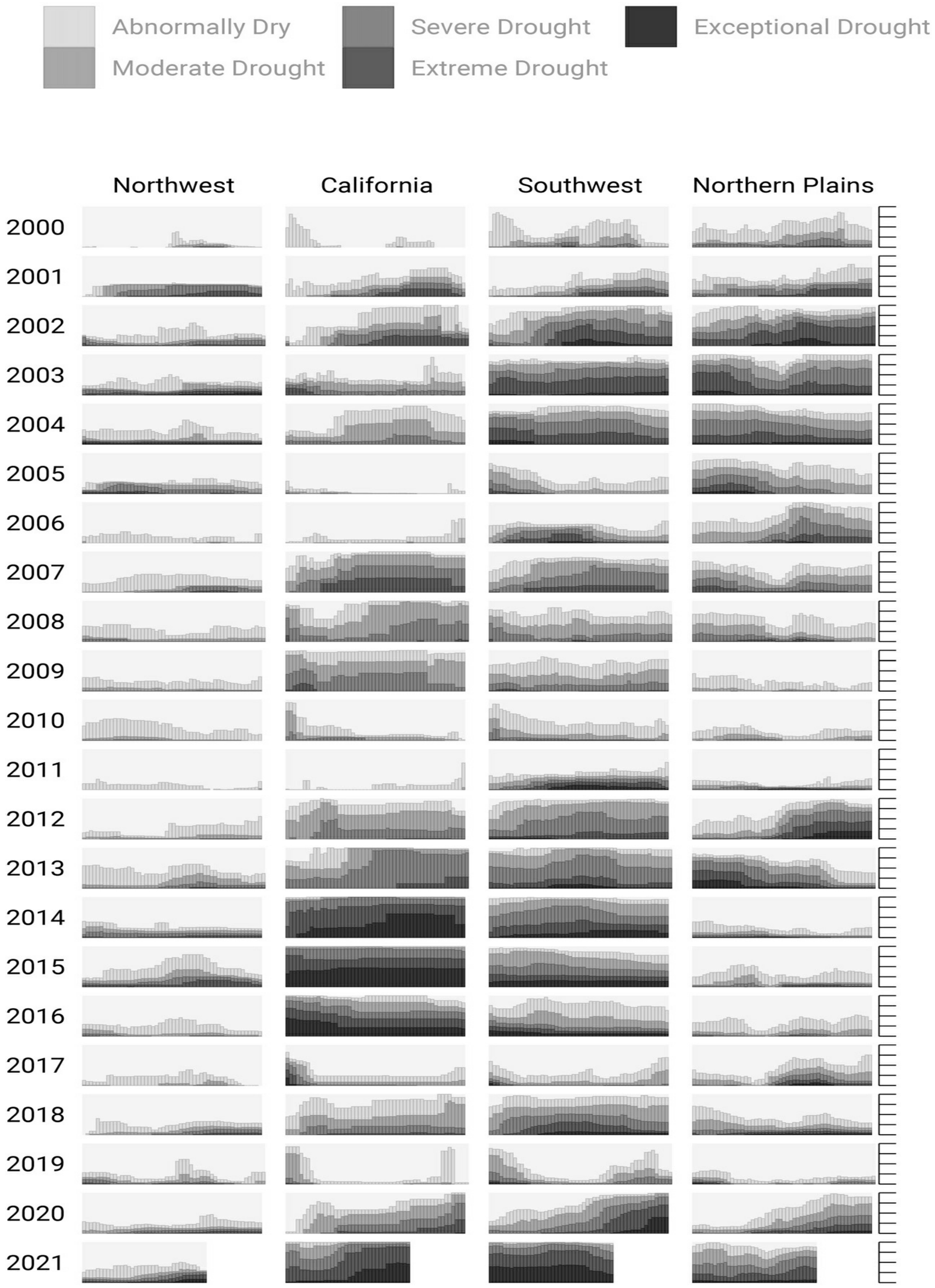
This chapter delves into why the data visualization that Scherer and Karamanis created is effective and introduces you to the *grammar of graphics*, a theory to make sense of graphs that underlies the `ggplot2` package. You’ll then learn how to use `ggplot2` by re-creating the drought graph step-by-step. In the process, I’ll highlight some key principles of high-quality data visualization that you can use to improve your own work.

The Drought Visualization

Other news organizations had relied on the same National Drought Center data in their stories, but Scherer and Karamanis visualized it so that it both grabs attention and communicates the scale of the phenomenon. Figure 2-1 shows a section of the final visualization (due to space constraints, I could include only four regions). The graph makes apparent the increase in drought conditions over the last two decades, especially in California and the Southwest.

To understand why this visualization is effective, let’s break it down. At the broadest level, the data visualization is notable for its minimalist aesthetic. For example, there are no grid lines and few text labels, as well as minimal text along the axes. Scherer and Karamanis removed what statistician Edward Tufte, in his 1983 book *The Visual Display of Quantitative Information* (Graphics Press), calls *chartjunk*. Tufte wrote that extraneous elements often hinder, rather than help, our understanding of charts (and researchers and data visualization designers have generally agreed).

Need proof that Scherer and Karamanis’s decluttered graph is better than the alternative? Figure 2-2 shows a version with a few tweaks to the code to include grid lines and text labels on axes.



Abnormally Dry
Moderate Drought
Severe Drought
Extreme Drought
Exceptional Drought
Northwest
California
Southwest
Northern Plains
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021

Figure 2-1: A section of the final drought visualization, with a few tweaks made to fit this book

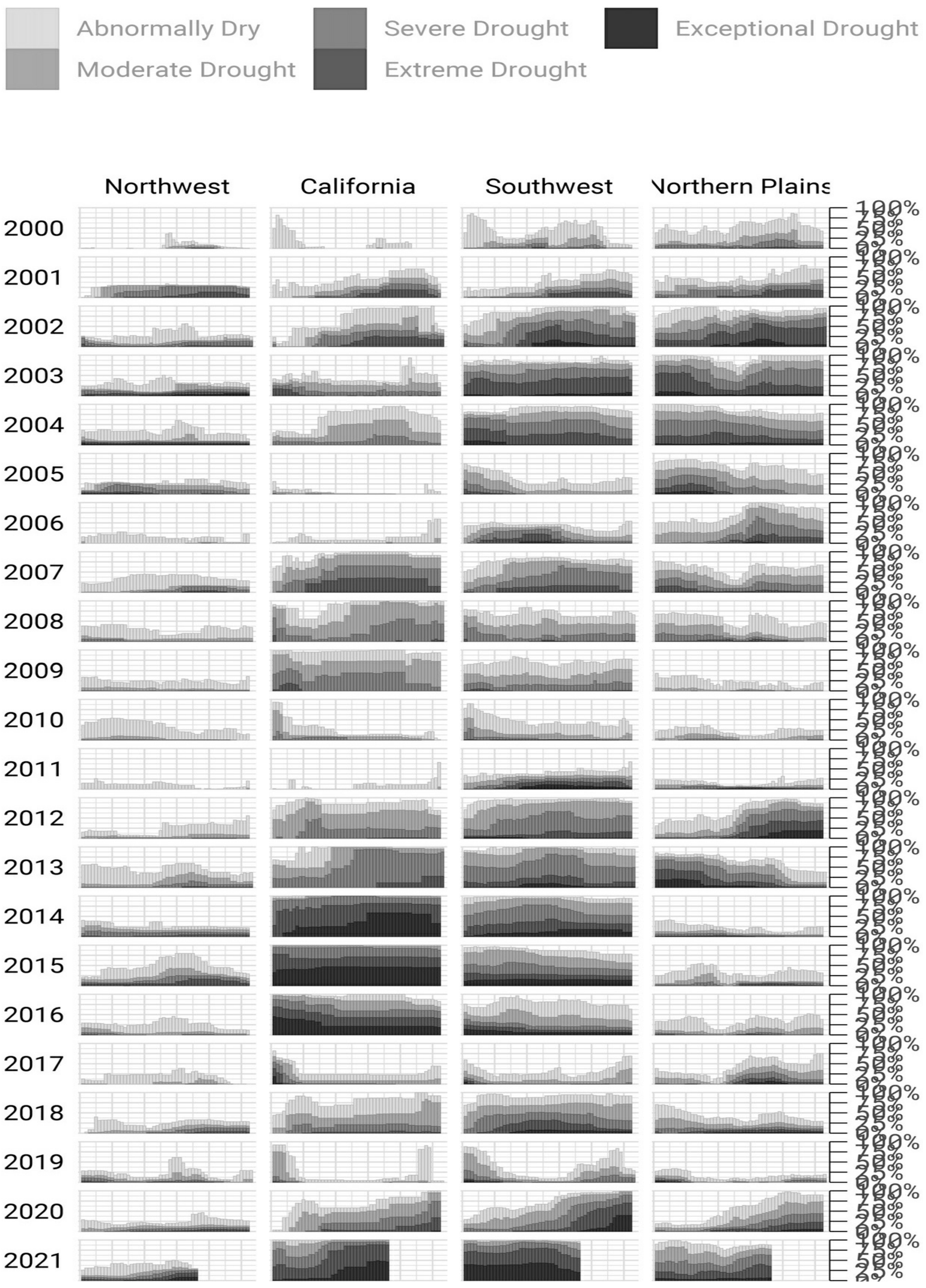

Abnormally Dry
Moderate Drought
Severe Drought
Extreme Drought
Exceptional Drought
Northwest
California
Southwest
Northern Plains
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
2010
2011
2012
2013
2014
2015
2016
2017
2018
2019
2020
2021
100%
75%
50%
25%
0%

Figure 2-2: The cluttered version of the drought visualization

It’s not just that this cluttered version looks worse; the clutter actively inhibits understanding. Rather than focusing on overall drought patterns (the point of the graph), our brains get stuck reading repetitive and unnecessary axis text.

One of the best ways to reduce clutter is to break a single chart into a set of component charts, as Scherer and Karamanis have done (this approach, known as *faceting*, will be discussed further in “Faceting the Plot” on page 36). Each rectangle represents one region in one year. Filtering the larger chart to show the Southwest region in 2003 produces the graph shown in Figure 2-3, where the x-axis indicates the week and the y-axis indicates the percentage of that region at different drought levels.

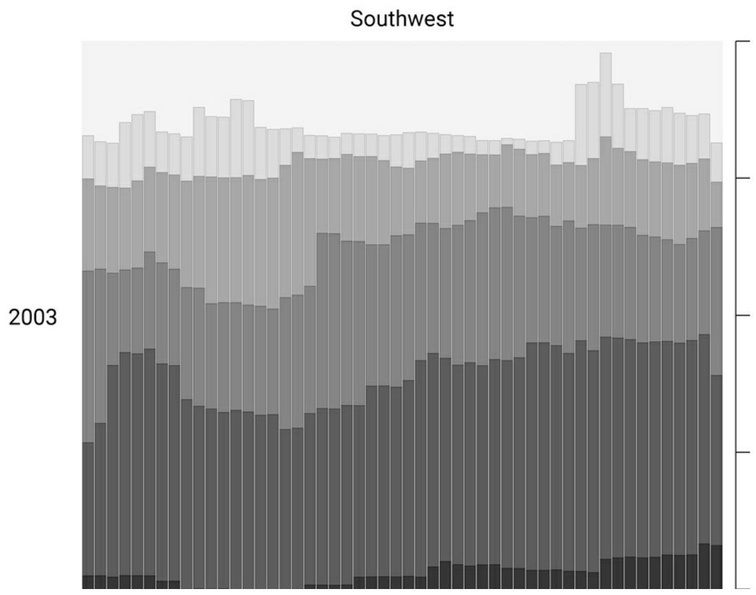


Figure 2-3: A drought visualization for the Southwest in 2003

Zooming in on a single region in a single year also makes the color choices more obvious. The lightest orange bars (lightest gray as printed here) show the percentage of the region that is abnormally dry, and the darkest purple bars (darkest gray as printed) show the percentage experiencing exceptional drought conditions. As you’ll see shortly, this range of colors was intentionally chosen to make differences in the drought levels visible to all readers.

Despite the graph’s complexity, the R code that Scherer and Karamanis wrote to produce it is relatively simple, due largely to a theory called the *grammar of graphics*.

The Grammar of Graphics

When working in Excel, you begin by selecting the type of graph you want to make. Need a bar chart? Click the bar chart icon. Need a line chart? Click the line chart icon. If you’ve only ever made charts in Excel, this first step may seem so obvious that you’ve never even given the data visualization process much thought, but in fact there are many ways to think about graphs. For example, rather than thinking of graph types as distinct, we can recognize and use their commonalities as the starting point for making them.

This approach to thinking about graphs comes from the late statistician Leland Wilkinson. For years, Wilkinson thought deeply about what data visualization is and how we can describe it. In 1999 he published a book called *The Grammar of Graphics* (Springer) that sought to develop a consistent way of describing all graphs. In it, Wilkinson argued that we should think of plots not as distinct types, à la Excel, but as following a grammar that we can use to describe *any* plot. Just as English grammar tells us that a noun is typically followed by a verb (which is why “he goes” works, while the opposite, “goes he,” does not), the grammar of graphics helps us understand why certain graph types “work.”

Thinking about data visualization through the lens of the grammar of graphics helps highlight, for example, that graphs typically have some data that is plotted on the x-axis and other data that is plotted on the y-axis. This is the case whether the graph is a bar chart or a line chart, as Figure 2-4 shows.

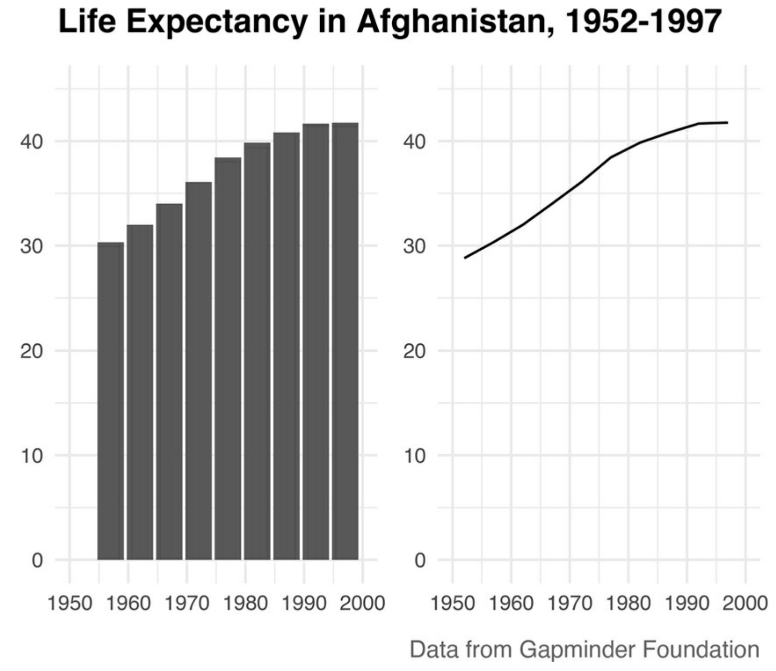


Figure 2-4: A bar chart and a line chart showing identical data

While the graphs look different (and would, to the Excel user, be different types of graphs), Wilkinson's grammar of graphics emphasizes their similarities. (Incidentally, Wilkinson's feelings on graph-making tools like Excel became clear when he wrote that "most charting packages channel user requests into a rigid array of chart types.")

When Wilkinson wrote his book, no data visualization tool could implement his grammar of graphics. This would change in 2010, when Hadley Wickham announced the `ggplot2` package for R in the article "A Layered Grammar of Graphics," published in the *Journal of Computational and Graphical Statistics*. By providing the tools to implement Wilkinson's ideas, `ggplot2` would come to revolutionize the world of data visualization.

Working with ggplot

The `ggplot2` R package (which I, like nearly everyone in the data visualization world, will refer to simply as *ggplot*) relies on the idea of plots having multiple layers. This section will walk you through some of the most important ones. You'll begin by selecting variables to map to aesthetic properties. Then you'll choose a geometric object to use to represent your data. Next, you'll change the aesthetic properties of your chart (its color scheme, for example) using a `scale_` function. Finally, you'll use a `theme_` function to set the overall look and feel of your plot.

Mapping Data to Aesthetic Properties

To create a graph with ggplot, you begin by mapping data to aesthetic properties. All this really means is that you use elements like the x- or y-axis, color, and size (the so-called *aesthetic properties*) to represent variables. You'll use the data on life expectancy in Afghanistan, introduced in Figure 2-4, to generate a plot. To access this data, enter the following code:

```
library(tidyverse)

gapminder_10_rows <- read_csv("https://data.rfortherestofus.c
om/data/gapminder_10_rows.csv")
```

This code first loads the `tidyverse` package, introduced in Chapter 1, and then uses the `read_csv()` function to access data from the book's website and assign it to the `gapminder_10_rows` object.

The resulting `gapminder_10_rows` tibble looks like this:

```
#> # A tibble: 10 × 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <fct>       <fct>     <int>   <dbl>    <int>     <dbl>
#>  1 Afghanistan Asia       1952    28.8  8425333      779.
#>  2 Afghanistan Asia       1957    30.3  9240934      821.
#>  3 Afghanistan Asia       1962    32.0 10267083      853.
#>  4 Afghanistan Asia       1967    34.0 11537966      836.
#>  5 Afghanistan Asia       1972    36.1 13079460      740.
#>  6 Afghanistan Asia       1977    38.4 14880372      786.
#>  7 Afghanistan Asia       1982    39.9 12881816      978.
#>  8 Afghanistan Asia       1987    40.8 13867957      852.
#>  9 Afghanistan Asia       1992    41.7 16317921      649.
#> 10 Afghanistan Asia       1997    41.8 22227415      635.
```

This output is a shortened version of the full `gapminder` data frame, which includes over 1,700 rows of data.

Before making a chart with ggplot, you need to decide which variable to put on the x-axis and which to put on the y-axis. For data showing change over time, it’s common to put the date (in this case, `year`) on the x-axis and the changing value (in this case, `lifeExp`) on the y-axis. To do so, define the `ggplot()` function as follows:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp
  )
)
```

This function contains numerous arguments. Each argument goes on its own line, for the sake of readability, separated by commas. The `data` argument tells R to use the data frame `gapminder_10_rows`, and the `mapping` argument maps `year` to the x-axis and `lifeExp` to the y-axis.

Running this code produces the chart in Figure 2-5, which doesn’t look like much yet.

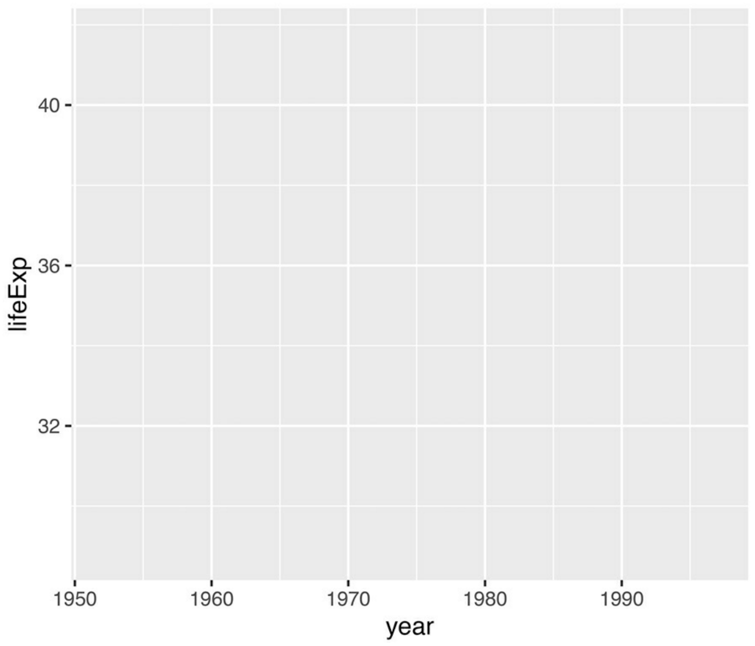


Figure 2-5: A blank chart that maps year values to the x-axis and life expectancy values to the y-axis

Notice that the x-axis corresponds to `year` and the y-axis corresponds to `lifeExp`, and the values on both axes match the scope of the data. In the `gapminder_10_rows` data frame, the first year is 1952 and the last year is 1997. The range of the x-axis has been created with this data in mind. Likewise, the values for `lifeExp`, which go from about 28 to about 42, will fit nicely on the y-axis.

Choosing the Geometric Objects

Axes are nice, but the graph is missing any type of visual representation of the data. To get this, you need to add the next ggplot layer: geoms. Short for *geometric objects*, *geoms* are functions that provide different ways of representing data. For example, to add points to the graph, you use `geom_point()`:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp
  )
) +
  geom_point()
```

Now the graph shows that people in 1952 had a life expectancy of about 28 and that this value rose every year in the dataset (see Figure 2-6).

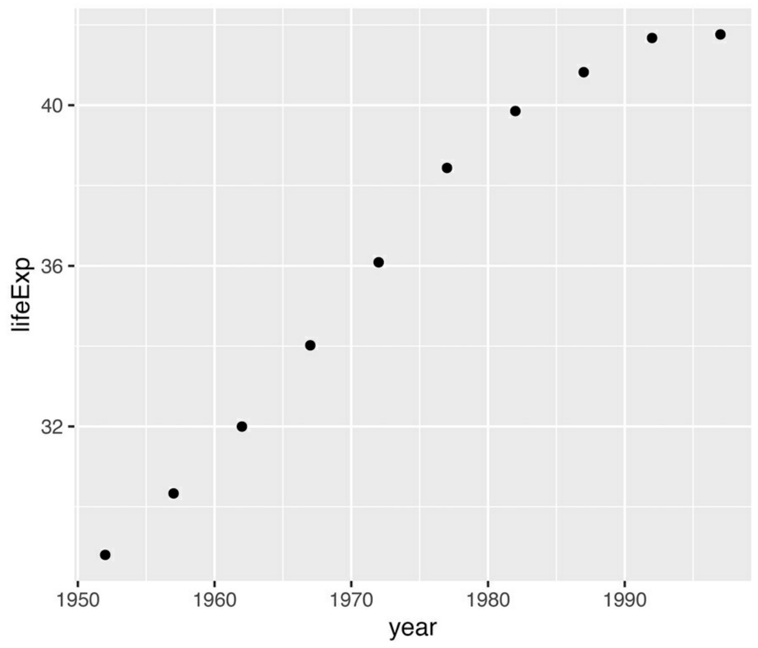


Figure 2-6: The life expectancy chart with points added

Say you change your mind and want to make a line chart instead. All you have to do is replace `geom_point()` with `geom_line()` like so:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp
  )
) +
  geom_line()
```

Figure 2-7 shows the result.

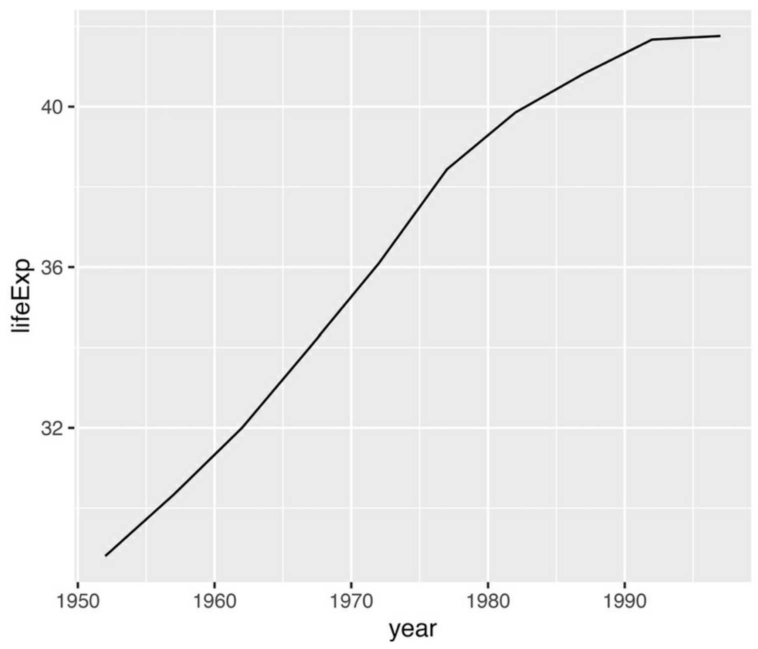


Figure 2-7: The same data as a line chart

To really get fancy, you could add both `geom_point()` and `geom_line()` as follows:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp
  )
) +
  geom_point() +
  geom_line()
```

This code generates a line chart with points, as shown in Figure 2-8.

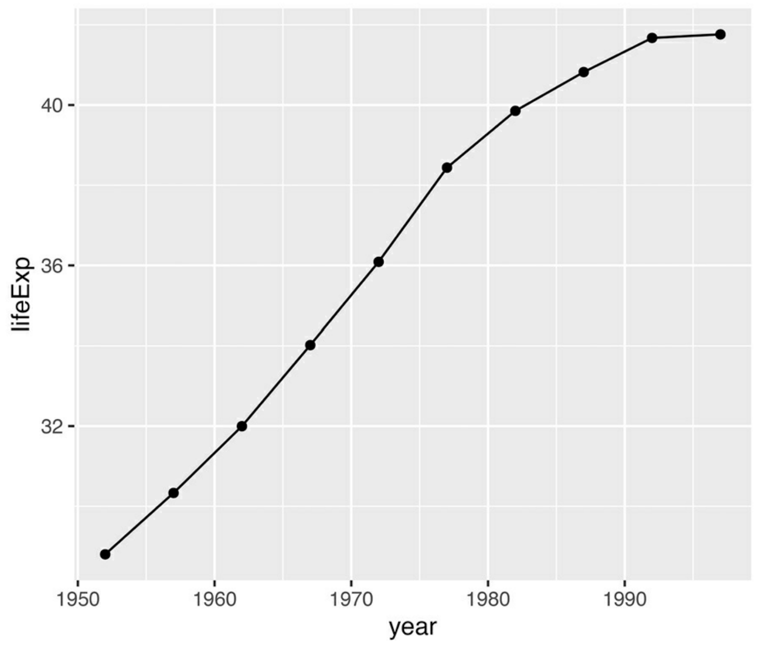


Figure 2-8: The same data with both points and a line

You can swap in `geom_col()` to create a bar chart:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp
  )
) +
  geom_col()
```

Notice in Figure 2-9 that the y-axis range has been automatically updated, going from 0 to 40 to account for the different geom.

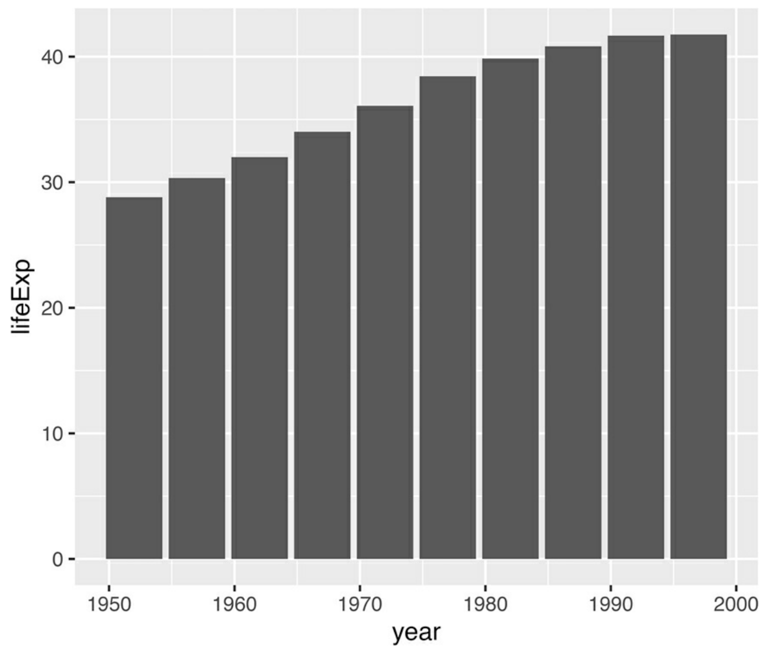


Figure 2-9: The life expectancy data as a bar chart

As you can see, the difference between a line chart and a bar chart isn't as great as the Excel chart-type picker might have you believe. Both can have the same underlying properties (namely, years on the x-axis and life expectancies on the y-axis). They simply use different geometric objects to visually represent the data.

Many geoms are built into ggplot. In addition to `geom_bar()`, `geom_point()`, and `geom_line()`, the geoms `geom_histogram()`, `geom_boxplot()`, and `geom_area()` are among the most commonly used. To see all geoms, visit the ggplot documentation website at *https://ggplot2.tidyverse.org/reference/index.xhtml#geoms*.

Altering Aesthetic Properties

Before we return to the drought data visualization, let's look at a few additional layers you can use to alter the bar chart. Say you want to change the color of the bars. In the grammar of graphics approach to chart-making, this means mapping some variable to the aesthetic property of `fill`. (For a bar chart, the aesthetic property of `color` would change only the outline of each bar.) In the same way that you mapped `year` to the x-axis and `lifeExp` to the y-axis, you can map `fill` to a variable, such as `year`:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp,
    fill = year
  )
) +
  geom_col()
```

Figure 2-10 shows the result. Now the fill is darker for earlier years and lighter for later years (as also indicated by the legend, added to the right of the plot).

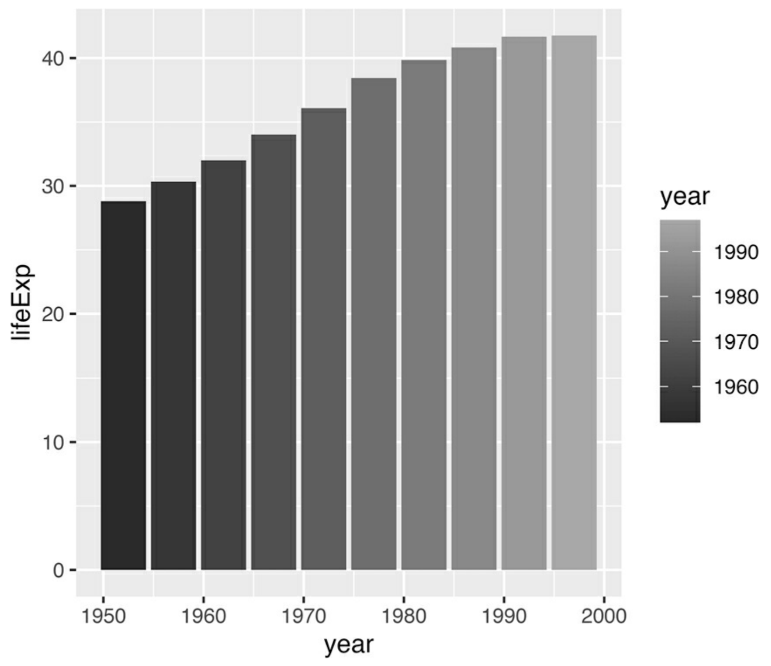


Figure 2-10: The same chart, now with added colors

To change the fill colors, use a new scale layer with the `scale_fill_viridis_c()` function (the `c` at the end of the function name refers to the fact that the data is continuous, meaning it can take any numeric value):

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp,
    fill = year
  )
) +
  geom_col() +
  scale_fill_viridis_c()
```

This function changes the default palette to one that is colorblind-friendly and prints well in grayscale. The `scale_fill_viridis_c()` function is just one of many that start with `scale_` and can alter the fill scale. Chapter 11 of *ggplot2: Elegant Graphics for Data Analysis*, 3rd edition, discusses various color and fill scales. You can read it online at *https://ggplot2-book.org/scales-colour.xhtml*.

Setting a Theme

The final layer we'll look at is the theme layer, which allows you to change the overall look and feel of your plots (including their background and grid lines). As with the `scale_` functions, a number of functions also start with `theme_`. Add `theme_minimal()` as follows:

```
ggplot(
  data = gapminder_10_rows,
  mapping = aes(
    x = year,
    y = lifeExp,
    fill = year
  )
) +
  geom_col() +
  scale_fill_viridis_c() +
  theme_minimal()
```

This theme starts to declutter the plot, as you can see in Figure 2-11.

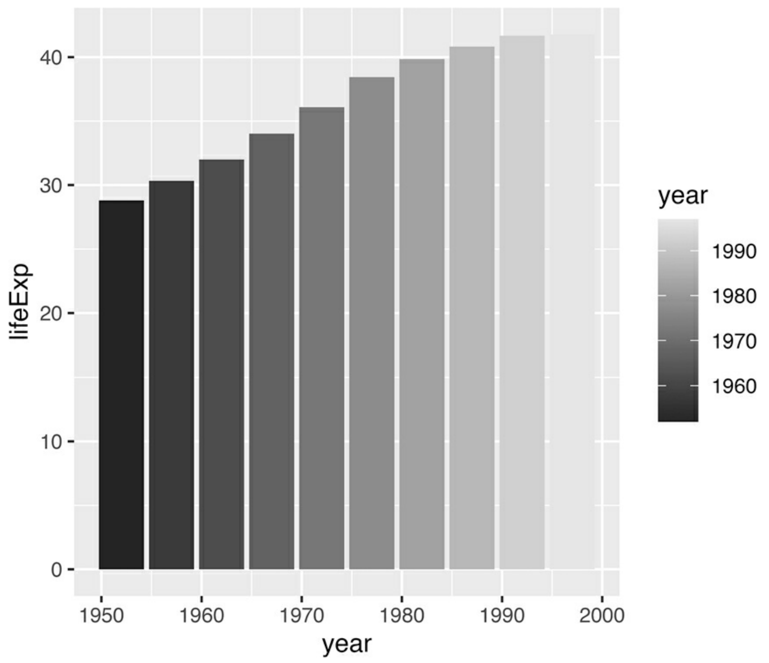


Figure 2-11: The same chart with theme_minimal() added

By now, you should see why Hadley Wickham described the `ggplot2` package as using a layered grammar of graphics. It implements Wilkinson's theory by creating multiple layers: first, variables to map to aesthetic properties; second, geoms to represent the data; third, the `scale_` function to adjust aesthetic properties; and finally, the `theme_` function to set the plot's overall look and feel.

You could still improve this plot in many ways, but instead let's return to the drought data visualization by Scherer and Karamanis. By walking through their code, you'll learn about making high-quality data visualization with ggplot and R.

Re-creating the Drought Visualization

The drought visualization code relies on a combination of ggplot fundamentals and some lesser-known tweaks that make it really shine. To understand how Scherer and Karamanis made their data visualization, we'll start with a simplified version of their code, then build it up layer by layer, adding elements as we go.

First, you'll import the data. Scherer and Karamanis did a bunch of data wrangling on the raw data, but I've saved the simplified output for you.

Because it’s in JavaScript Object Notation (JSON) format, Scherer and Karamanis use the `import()` function from the `rio` package, which simplifies the process of importing JSON data:

```
library(rio)

dm_perc_cat_hubs <- import("https://data.rfortherestofus.com/
dm_perc_cat_hubs.json"))
```

JSON is a common format for data used in web applications, though it’s far less common in R, where it can be complicated to work with. Luckily, the `rio` package simplifies its import.

Plotting One Region and Year

Scherer and Karamanis’s final plot consists of many years and regions. To see how they created it, we’ll start by looking at just the Southwest region in 2003.

First, you need to create a data frame. You’ll use the `filter()` function twice: the first time to keep only data for the Southwest region, and the second time to keep only data from 2003. In both cases, you use the following syntax:

```
filter(variable_name == value)
```

This tells R to keep only observations where *`variable_name`* is equal to some *`value`*. The code starts with the `dm_perc_cat_hubs_raw` data frame before filtering it and then saving it as a new object called `southwest_2003`:

```
southwest_2003 <- dm_perc_cat_hubs %>%
  filter(hub == "Southwest") %>%
  filter(year == 2003)
```

To take a look at this object and see the variables you have to work with, enter **`southwest_2003`** in the console, which should return this output:

```
#> # A tibble: 255 × 7
#>    date       hub  category  percentage  year  week max_we
```

```
ek
#>    <date>    <fct> <fct>          <dbl> <dbl> <dbl>    <db
l>
#>  1 2003-12-30 Sout... D0           0.0718  2003    52
  52
#>  2 2003-12-30 Sout... D1           0.0828  2003    52
  52
#>  3 2003-12-30 Sout... D2           0.2693  2003    52
  52
#>  4 2003-12-30 Sout... D3           0.3108  2003    52
  52
#>  5 2003-12-30 Sout... D4           0.0796  2003    52
  52
#>  6 2003-12-23 Sout... D0           0.0823  2003    51
  52
#>  7 2003-12-23 Sout... D1           0.1312  2003    51
  52
#>  8 2003-12-23 Sout... D2           0.1886  2003    51
  52
#>  9 2003-12-23 Sout... D3           0.3822  2003    51
  52
#> 10 2003-12-23 Sout... D4           0.0828  2003    51
  52
#> # 245 more rows
```

The `date` variable represents the start date of the week in which the observation took place. The `hub` variable is the region, and `category` is the level of drought: a value of `D0` indicates the lowest level of drought, while `D5` indicates the highest level. The `percentage` variable is the percentage of that region in that drought category, ranging from `0` to `1`. The `year` and `week` variables are the observation year and week number (beginning with week 1). The `max_week` variable is the maximum number of weeks in a given year.

Now you can use this `southwest_2003` object for your plot:

```
ggplot(
  data = southwest_2003,
  aes(
    x = week,
    y = percentage,
    fill = category
  )
) +
  geom_col()
```

The `ggplot()` function tells R to put `week` on the x-axis and `percentage` on the y-axis, as well as to use the `category` variable for the `fill` color. The `geom_col()` function creates a bar chart in which each bar's `fill` color represents the percentage of the region at each drought level for that particular week, as shown in Figure 2-12.

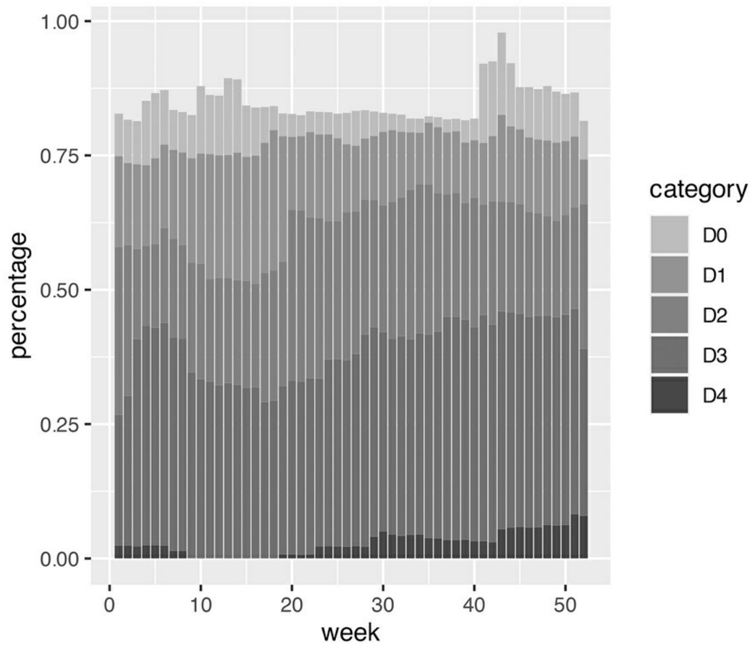


Figure 2-12: One year (2003) and region (Southwest) of the drought visualization

The colors, which include bright pinks, blues, greens, and reds (displayed in grayscale here), don't match the final version of the plot, but you can start to see the outlines of Scherer and Karamanis's data visualization.

Changing Aesthetic Properties

Scherer and Karamanis next selected different `fill` colors for their bars. To do so, they used the `scale_fill_viridis_d()` function. The *d* here means that the data to which the fill scale is being applied has discrete categories (D0, D1, D2, D3, D4, and D5):

```
ggplot(
  data = southwest_2003,
  aes(
    x = week,
    y = percentage,
    fill = category
  )
) +
  geom_col() +
  scale_fill_viridis_d(
    option = "rocket",
    direction = -1
  )
```

They used the argument `option = "rocket"` to select the rocket palette, whose colors range from cream to nearly black. You could use several other palettes within the `scale_fill_viridis_d()` function; see them at *https://sjmgarnier.github.io/viridisLite/reference/viridis.xhtml*.

Then they used the `direction = -1` argument to reverse the order of fill colors so that darker colors mean higher drought conditions.

Scherer and Karamanis also tweaked the appearance of the x- and y-axes:

```
ggplot(
  data = southwest_2003,
  aes(
    x = week,
    y = percentage,
    fill = category
  )) +
  geom_col() +
  scale_fill_viridis_d(
    option = "rocket",
    direction = -1
  ) +
  scale_x_continuous(
    name = NULL,
    guide = "none"
  ) +
  scale_y_continuous(
    name = NULL,
    labels = NULL,
    position = "right"
  )
```

On the x-axis, they removed both the axis title ("week") using `name = NULL` and the axis labels (the weeks numbered 0 to 50) with `guide = "none"`. On the y-axis, they removed the title and text showing percentages using `labels = NULL`, which functionally does the same thing as `guide = "none"`. They also moved the axis lines themselves to the right side using `position = "right"`. These axis lines are apparent only as tick marks at this point but will become more visible later. Figure 2-13 shows the result of these tweaks.

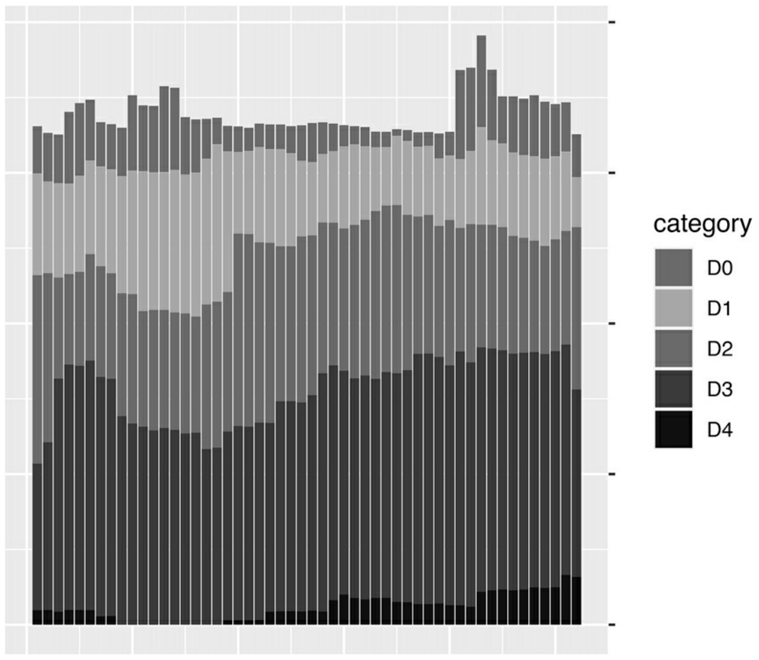


Figure 2-13: The 2003 drought data for the Southwest with adjustments to the x- and y-axes

Up to this point, we've focused on one of the single plots that make up the larger data visualization. But the final product that Scherer and Karamanis made is actually 176 plots visualizing 22 years and 8 regions. Let's discuss the ggplot feature they used to create all of these plots.

Faceting the Plot

One of ggplot's most useful capabilities is *faceting* (or, as it's more commonly known in the data visualization world, *small multiples*). Faceting uses a variable to break down a single plot into multiple plots. For example, think of a line chart showing life expectancy by country over time; instead of multiple lines on one plot, faceting would create multiple plots with one line per plot. To specify which variable to put in the rows and which to put in the columns of your faceted plot, you use the `facet_grid()` function, as Scherer and Karamanis did in their code:

```
dm_perc_cat_hubs %>%
  filter(hub %in% c(
    "Northwest",
    "California",
    "Southwest",
    "Northern Plains"
  )) %>%
  ggplot(aes(
    x = week,
    y = percentage,
    fill = category
  )) +
  geom_col() +
  scale_fill_viridis_d(
    option = "rocket",
    direction = -1
  ) +
  scale_x_continuous(
    name = NULL,
    guide = "none"
  ) +
  scale_y_continuous(
    name = NULL,
    labels = NULL,
    position = "right"
  ) +
  facet_grid(
    rows = vars(year),
    cols = vars(hub),
    switch = "y"
  )
```

Scherer and Karamanis put `year` in rows and `hub` (region) in columns. The `switch = "y"` argument moves the year label from the right side (where it appears by default) to the left. With this code in place, you can see the final plot coming together in Figure 2-14.

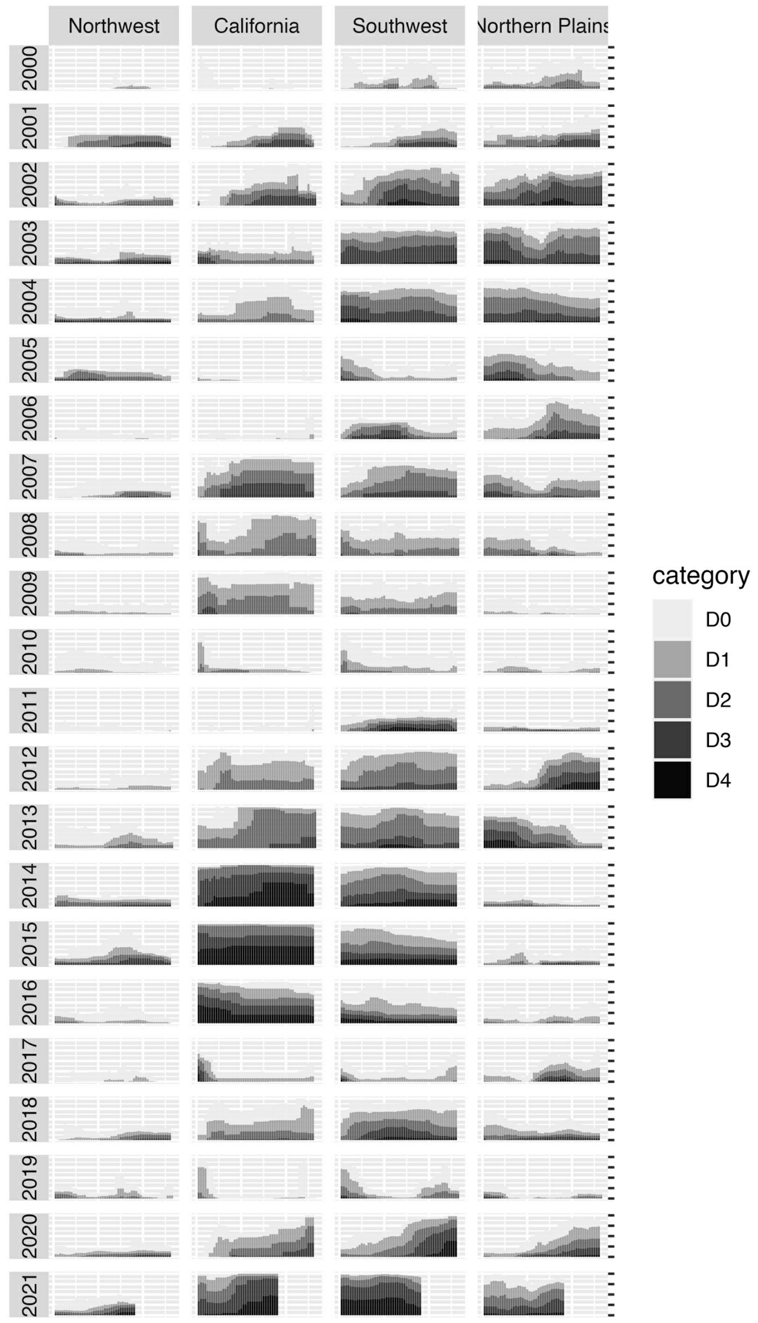


Figure 2-14: The faceted version of the drought visualization

Incredibly, the broad outlines of the plot took just 10 lines of code to create. The rest of the code falls into the category of small polishes. That's not to minimize how important small polishes are (very) or the time it takes to create them (a lot). It does show, however, that a little bit of ggplot goes a long way.

Adding Final Polishes

Now let's look at a few of the small polishes that Scherer and Karamanis made. The first is to apply a theme. They used `theme_light()`, which removes the default gray background and changes the font to Roboto using the `base_family` argument.

The `theme_light()` function is what's known as a *complete theme*, one that changes the overall look and feel of a plot. The ggplot package has multiple complete themes that you can use (they're listed at *https://ggplot2.tidyverse.org/reference/index.xhtml#themes*). Individuals and organizations also make their own themes, as you'll do in Chapter 3. For a discussion of which themes you might consider using, see my blog post at *https://rfortherestofus.com/2019/08/themes-to-improve-your-ggplot-figures*.

Scherer and Karamanis didn't stop by simply applying `theme_light()`. They also used the `theme()` function to make additional tweaks to the plot's design:

```
dm_perc_cat_hubs %>%
  filter(hub %in% c(
    "Northwest",
    "California",
    "Southwest",
    "Northern Plains"
  )) %>%
  ggplot(aes(
    x = week,
    y = percentage,
    fill = category
  )) +
  geom_rect(
    aes(
      xmin = .5,
      xmax = max_week + .5,
      ymin = -0.005,
      ymax = 1
    ),
    fill = "#f4f4f9",
    color = NA,
    size = 0.4
  ) +
  geom_col() +
  scale_fill_viridis_d(
    option = "rocket",
    direction = -1
  ) +
  scale_x_continuous(
    name = NULL,
    guide = "none"
  ) +
  scale_y_continuous(
    name = NULL,
    labels = NULL,
    position = "right"
  ) +
  facet_grid(
    rows = vars(year),
    cols = vars(hub),
    switch = "y"
  ) +
  theme_light(base_family = "Roboto") +
  theme(
    axis.title = element_text(
      size = 14,
      color = "black"
    ),
    axis.text = element_text(
      family = "Roboto Mono",
      size = 11
    ),
  ❶ axis.line.x = element_blank(),
    axis.line.y = element_line(
      color = "black",
      size = .2
    ),
    axis.ticks.y = element_line(
      color = "black",
      size = .2
    ),
    axis.ticks.length.y = unit(2, "mm"),
  ❷ legend.position = "top",
    legend.title = element_text(
      color = "#2DAADA",
      face = "bold"
    ),
    legend.text = element_text(color = "#2DAADA"),
    strip.text.x = element_text(
      hjust = .5,
      face = "plain",
      color = "black",
      margin = margin(t = 20, b = 5)
    ),
    strip.text.y.left = element_text(
   ❸ angle = 0,
      vjust = .5,
      face = "plain",
      color = "black"
    ),
    strip.background = element_rect(
      fill = "transparent",
      color = "transparent"
    ),
  ❹ panel.grid.minor = element_blank(),
    panel.grid.major = element_blank(),
    panel.spacing.x = unit(0.3, "lines"),
    panel.spacing.y = unit(0.25, "lines"),
  ❺ panel.background = element_rect(
      fill = "transparent",
      color = "transparent"
    ),
    panel.border = element_rect(
      color = "transparent",
      size = 0
    ),
    plot.background = element_rect(
      fill = "transparent",
      color = "transparent",
      size = .4
    ),
    plot.margin = margin(rep(18, 4))
  )
  )
```

The code in the `theme()` function does many different things, but let's look at a few of the most important. First, it moves the legend from the right side (the default) to the top of the plot ❷. Then, the `angle = 0` argument rotates the year text in the columns from vertical to horizontal ❸. Without this argument, the years would be much less legible.

The `theme()` function also makes the distinctive axis lines and ticks that appear on the right side of the final plot ❶. Calling `element_blank()` removes all grid lines ❹. Finally, this code removes the borders and gives each individual plot a transparent background ❺.

You might be thinking, *Wait. Didn't the individual plots have a gray background behind them?* Yes, dear reader, they did. Scherer and Karamanis made these with a separate geom, `geom_rect()`:

```
geom_rect(
  aes(
    xmin = .5,
    xmax = max_week + .5,
    ymin = -0.005,
    ymax = 1
  ),
  fill = "#f4f4f9",
  color = NA,
  size = 0.4
)
```

They also set some additional aesthetic properties specific to this geom —`xmin`, `xmax`, `ymin`, and `ymax`—which determine the boundaries of the rectangle it produces. The result is a gray background behind each small multiple, as shown in Figure 2-15.

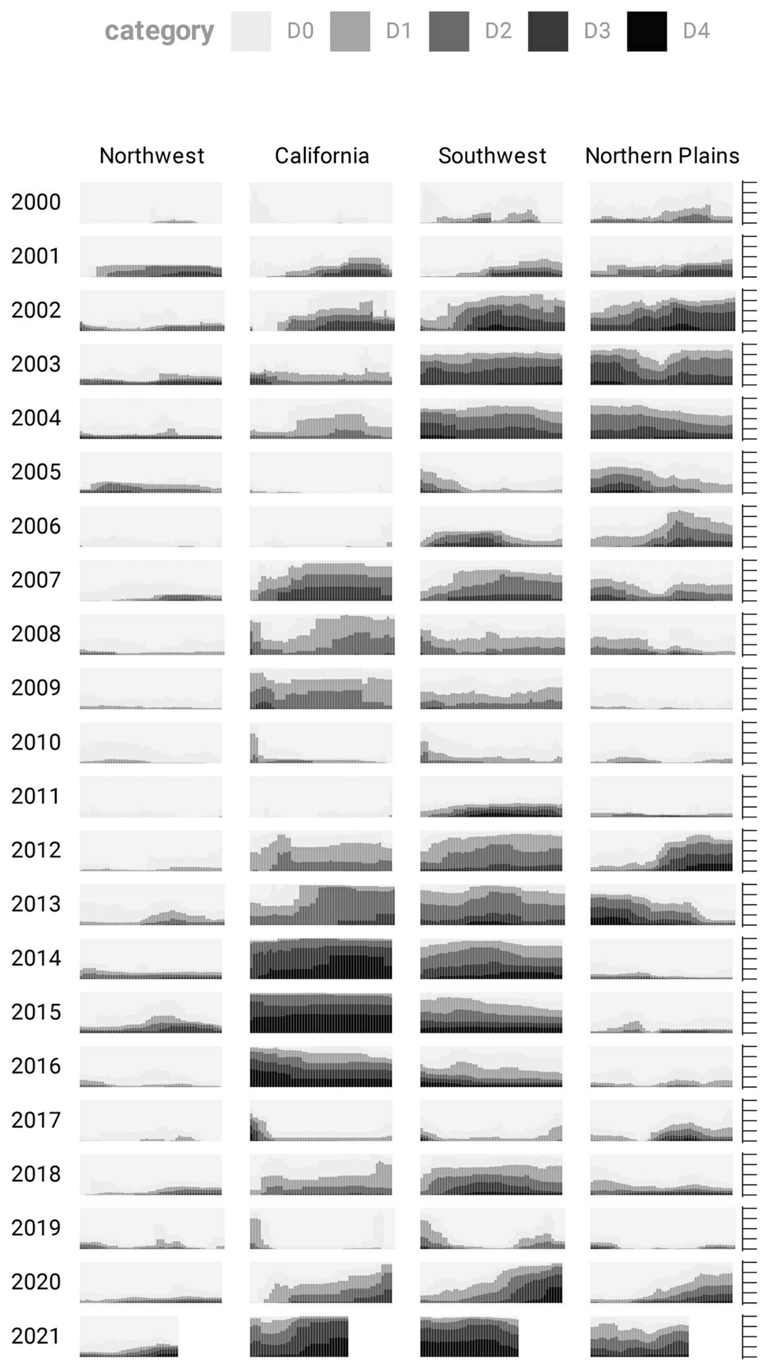


Figure 2-15: The faceted version of the drought visualization with a gray background behind each small multiple

Finally, Scherer and Karamanis made some tweaks to the legend. Previously you saw a simplified version of the `scale_fill_viridis_d()` function. Here's a more complete version:

```
scale_fill_viridis_d(
  option = "rocket",
  direction = -1,
  name = "Category:",
  labels = c(
    "Abnormally Dry",
    "Moderate Drought",
    "Severe Drought",
    "Extreme Drought",
    "Exceptional Drought"
  )
)
```

The `name` argument sets the legend title, and the `labels` argument specifies the labels that show up in the legend. Figure 2-16 shows the result of these changes.

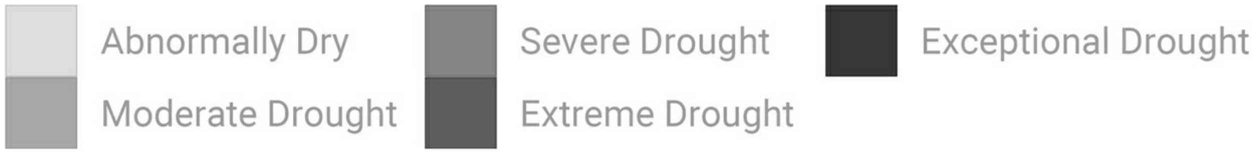


Figure 2-16: The drought visualization with changes to the legend text

Rather than D0, D1, D2, D3, and D4, the legend text now reads Abnormally Dry, Moderate Drought, Severe Drought, Extreme Drought, and Exceptional Drought—much more user-friendly categories.

The Complete Visualization Code

While I've shown you a nearly complete version of the code that Scherer and Karamanis wrote, I made some small changes to make it easier to understand. If you're curious, the full code is here:

```
ggplot(dm_perc_cat_hubs, aes(week, percentage)) +
  geom_rect(
    aes(
      xmin = .5,
      xmax = max_week + .5,
      ymin = -0.005,
      ymax = 1
    ),
    fill = "#f4f4f9",
    color = NA,
    size = 0.4,
    show.legend = FALSE
  ) +
  geom_col(
    aes(
      fill = category,
      fill = after_scale(addmix(
        darken(
          fill,
          .05,
          space = "HLS"
        ),
        "#d8005a",
        .15
      )),
      color = after_scale(darken(
        fill,
        .2,
        space = "HLS"
      ))
    ),
    width = .9,
    size = 0.12
  ) +
  facet_grid(
    rows = vars(year),
    cols = vars(hub),
    switch = "y"
  ) +
  coord_cartesian(clip = "off") +
  scale_x_continuous(
    expand = c(.02, .02),
    guide = "none",
    name = NULL
  ) +
  scale_y_continuous(
    expand = c(0, 0),
    position = "right",
    labels = NULL,
    name = NULL
  ) +
  scale_fill_viridis_d(
    option = "rocket",
    name = "Category:",
    direction = -1,
    begin = .17,
    end = .97,
    labels = c(
      "Abnormally Dry",
      "Moderate Drought",
      "Severe Drought",
      "Extreme Drought",
      "Exceptional Drought"
    )
  ) +
  guides(fill = guide_legend(
    nrow = 2,
    override.aes = list(size = 1)
  )) +
  theme_light(
    base_size = 18,
    base_family = "Roboto"
  ) +
  theme(
    axis.title = element_text(
      size = 14,
      color = "black"
    ),
    axis.text = element_text(
      family = "Roboto Mono",
      size = 11
    ),
    axis.line.x = element_blank(),
    axis.line.y = element_line(
      color = "black",
      size = .2
    ),
    axis.ticks.y = element_line(
      color = "black",
      size = .2
    ),
    axis.ticks.length.y = unit(2, "mm"),
    legend.position = "top",
    legend.title = element_text(
      color = "#2DAADA",
      size = 18,
      face = "bold"
    ),
    legend.text = element_text(
      color = "#2DAADA",
      size = 16
    ),
    strip.text.x = element_text(
      size = 16,
      hjust = .5,
      face = "plain",
      color = "black",
      margin = margin(t = 20, b = 5)
    ),
    strip.text.y.left = element_text(
      size = 18,
      angle = 0,
      vjust = .5,
      face = "plain",
      color = "black"
    ),
    strip.background = element_rect(
      fill = "transparent",
      color = "transparent"
    ),
    panel.grid.minor = element_blank(),
    panel.grid.major = element_blank(),
    panel.spacing.x = unit(0.3, "lines"),
    panel.spacing.y = unit(0.25, "lines"),
    panel.background = element_rect(
      fill = "transparent",
      color = "transparent"
    ),
    panel.border = element_rect(
      color = "transparent",
      size = 0
    ),
    plot.background = element_rect(
      fill = "transparent",
      color = "transparent",
      size = .4
    ),
    plot.margin = margin(rep(18, 4))
  )
```

There are a few additional tweaks to color and spacing, but most of the code reflects what you’ve seen so far.

Summary

You may be thinking that ggplot is the solution to all of your data visualization problems. And yes, you have a new hammer, but not everything is a nail. If you look at the version of the data visualization that appeared in *Scientific American* in November 2021, you’ll see that some of its annotations aren’t visible in our re-creation. That’s because they were added in post-production. While you could have found ways to create them in ggplot, it’s often not the best use of your time. Get yourself 90 percent of the way there with ggplot and then use Illustrator, Figma, or a similar tool to finish your work.

Even so, ggplot is a very powerful hammer, used to make plots that you’ve seen in the *New York Times*, FiveThirtyEight, the BBC, and other well-known news outlets. Although it’s not the only tool that can generate high-quality data visualizations, it makes the process straightforward. The graph by Scherer and Karamanis shows this in several ways:

- It strips away extraneous elements, such as grid lines, to keep the focus on the data itself. Complete themes such as `theme_light()` and the `theme()` function allowed Scherer and Karamanis to create a decluttered visualization that communicates effectively.
- It uses well-chosen colors. The `scale_fill_viridis_d()` function allowed them to create a color scheme that demonstrates differences between groups, is colorblind-friendly, and shows up well when printed in grayscale.
- It uses faceting to break down data from two decades and eight regions into a set of graphs that come together to create a single plot. With a single call to the `facet_grid()` function, Scherer and Karamanis created over 100 small multiples that the tool automatically combined into a single plot.

Learning to create data visualizations in ggplot involves a significant time investment. But the long-term payoff is even greater. Once you learn how ggplot works, you can look at others’ code and learn how to improve your own. By contrast, when you make a data visualization in Excel, the series of point-and-click steps disappears into the ether. To re-create a visualization you made last week, you'll need to remember the exact steps you used, and to make someone else's data visualization, you'll need them to write up their process for you.

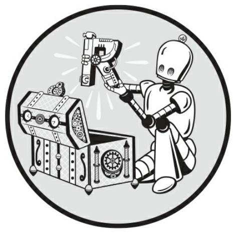
Because code-based data visualization tools allow you to keep a record of the steps you made, you don't have to be the most talented designer to make high-quality data visualizations with ggplot. You can study others' code, adapt it to your own needs, and create your own data visualization that not only is beautiful but also communicates effectively.

Additional Resources

- Will Chase, "The Glamour of Graphics," online course, accessed November 6, 2023, *https://rfortherestofus.com/courses/glamour/*.
- Kieran Healy, *Data Visualization: A Practical Introduction* (Princeton, NJ: Princeton University Press, 2018), *https://socviz.co*.
- Cédric Scherer, *Graphic Design with ggplot2* (Boca Raton, FL: CRC Press, forthcoming).
- Hadley Wickham, Danielle Navarro, and Thomas Lin Pedersen, *ggplot2: Elegant Graphics for Data Analysis*, 3rd ed. (New York: Springer, forthcoming), *https://ggplot2-book.org*.
- Claus Wilke, *Fundamentals of Data Visualization* (Sebastopol, CA: O'Reilly Media, 2019), *https://clauswilke.com/dataviz/*.

3

CUSTOM DATA VISUALIZATION THEMES



A *custom theme* is nothing more than a chunk of code that applies a set of small tweaks to all plots. So much of the work involved in making a professional chart consists of these kinds of adjustments. What font should you use? Where should the legend go? Should axes have titles? Should charts have grid lines? These questions may seem minor, but they have a major impact on the final product.

In 2018, BBC data journalists Nassos Stylianou and Clara Guibourg, along with their team, developed a custom ggplot theme that matches the BBC's style. By introducing this `bbplot` package for others to use, they changed their organization's culture, removed bottlenecks, and allowed the BBC to visualize data more creatively.

Rather than forcing everyone to copy the long code to tweak each plot they make, custom themes enable everyone who uses them to follow style guidelines and ensure that all data visualizations meet a brand's standards. For example, to understand the significance of the custom theme introduced at the BBC, it's helpful to know how things worked before `bbplot`. In the mid-2010s, journalists who wanted to make data visualization had two choices:

- Use an internal tool that could create data visualizations but was limited to the predefined charts it had been designed to generate.

- Use Excel to create mockups and then work with a graphic designer to finalize the charts. This approach led to better results and was much more flexible, but it required extensive, time-consuming back-and-forth with a designer.

Neither of these choices was ideal, and the BBC's data visualization output was limited. R freed the journalists from having to work with a designer. It wasn't that the designers were bad (they weren't), but ggplot allowed the journalists to explore different visualizations on their own. As the team improved their ggplot skills, they realized that it might be possible to produce more than just exploratory data visualizations and to create production-ready charts in R that could go straight onto the BBC website.

This chapter discusses the power of custom ggplot themes, then walks through the code in the `bbplot` package to demonstrate how custom themes work. You'll learn how to consolidate your styling code into a reusable function and how to consistently modify your plots' text, axes, grid lines, background, and other elements.

Styling a Plot with a Custom Theme

The `bbplot` package has two functions: `bbc_style()` and `finalise_plot()`. The latter deals with tasks like adding the BBC logo and saving plots in the correct dimensions. For now, let's look at the `bbc_style()` function, which applies a custom ggplot theme to make all the plots look consistent and follow BBC style guidelines.

An Example Plot

To see how this function works, you'll create a plot showing population data about several penguin species. You'll be using the `palmerpenguins` package, which contains data about penguins living on three islands in Antarctica. For a sense of what this data looks like, load the `palmerpenguins` and `tidyverse` packages:

```
library(palmerpenguins)

library(tidyverse)
```

Now you have data you can work with in an object called `penguins`. Here’s what the first 10 rows look like:

```
#> # A tibble: 344 × 8
#>    species island    bill_le... bill_...  flipp...  body_..
.    sex
#>    <fct>   <fct>          <dbl>    <dbl>     <int>     <int
>    <fct>
#>  1 Adelie  Torgersen      39.1     18.7       181       375
0    male
#>  2 Adelie  Torgersen      39.5     17.4       186       380
0    fema...
#>  3 Adelie  Torgersen      40.3       18       195       325
0    fema...
#>  4 Adelie  Torgersen        NA       NA        NA         N
A    <NA>
#>  5 Adelie  Torgersen      36.7     19.3       193       345
0    fema...
#>  6 Adelie  Torgersen      39.3     20.6       190       365
0    male
#>  7 Adelie  Torgersen      38.9     17.8       181       362
5    fema...
#>  8 Adelie  Torgersen      39.2     19.6       195       467
5    male
#>  9 Adelie  Torgersen      34.1     18.1       193       347
5    <NA>
#> 10 Adelie  Torgersen      42       20.2       190       425
0    <NA>
--snip--
```

To get the data in a more usable format, you’ll count how many penguins live on each island with the `count()` function from the `dplyr` package (one of several packages that are loaded with the `tidyverse`):

```
penguins %>%
  count(island)
```

This gives you some simple data that you can use for plotting:

```
#> # A tibble: 3 × 2
#>   island        n
#>   <fct>     <int>
#> 1 Biscoe      168
#> 2 Dream       124
#> 3 Torgersen    52
```

You’ll use this data multiple times in the chapter, so save it as an object called `penguins_summary` like so:

```
penguins_summary <- penguins %>%
  count(island)
```

Now you’re ready to create a plot. Before you see what `bbplot` does, make a plot with the ggplot defaults:

```
penguins_plot <- ggplot(
  data = penguins_summary,
  aes(
    x = island,
    y = n,
    fill = island
  )
) +
  geom_col() +
  labs(
    title = "Number of Penguins",
    subtitle = "Islands are in Antarctica",
    caption = "Data from palmerpenguins package"
  )
```

This code tells R to use the `penguins_summary` data frame, putting the island on the x-axis and the count of the number of penguins (`n`) on the y-axis, and making each bar a different color with the `fill` aesthetic property. Since you’ll modify this plot multiple times, saving it as an object called `penguins_plot` simplifies the process. Figure 3-1 shows the resulting plot.

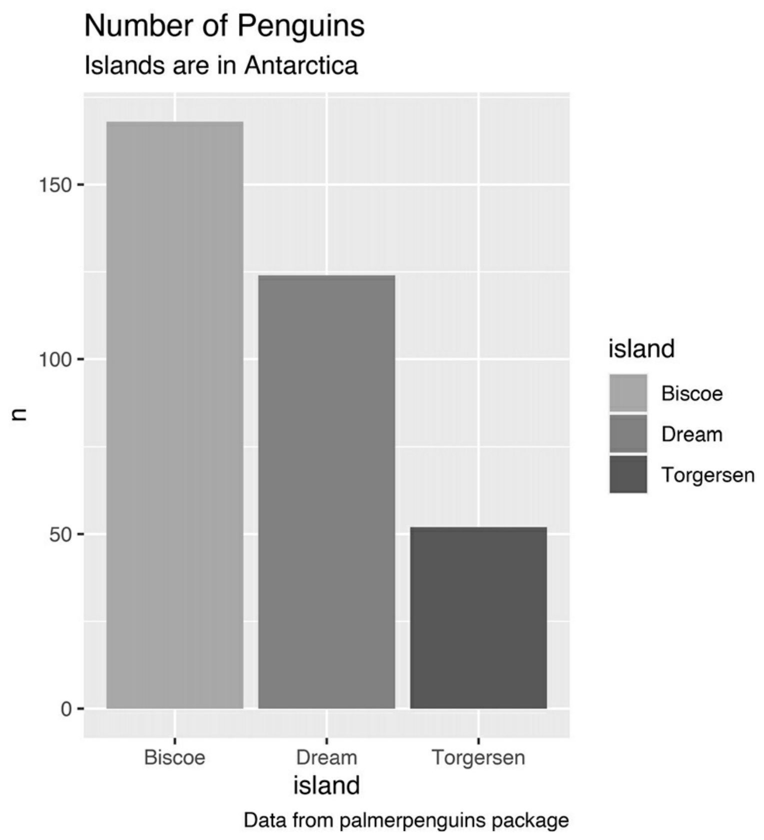


Figure 3-1: A chart with the default theme

This isn't the most aesthetically pleasing chart. The gray background is ugly, the y-axis title is hard to read because it's angled, and the text size overall is quite small. But don't worry, you'll be improving it soon.

The BBC's Custom Theme

Now that you have a basic plot to work with, you'll start making it look like a BBC chart. To do this, you need to install the `bbplot` package. First, install the `remotes` package using **`install.packages("remotes")`** so that you can access packages from remote sources. Then, run the following code to install `bbplot` from the GitHub repository at *https://github.com/bbc/bbplot* :

```
library(remotes)

install_github("bbc/bbplot")
```

Once you've installed the `bbplot` package, load it and apply the `bbc_style()` function to the `penguins_plot` as follows:

```
library(bbplot)

penguins_plot +
  bbc_style()
```

Figure 3-2 shows the result.

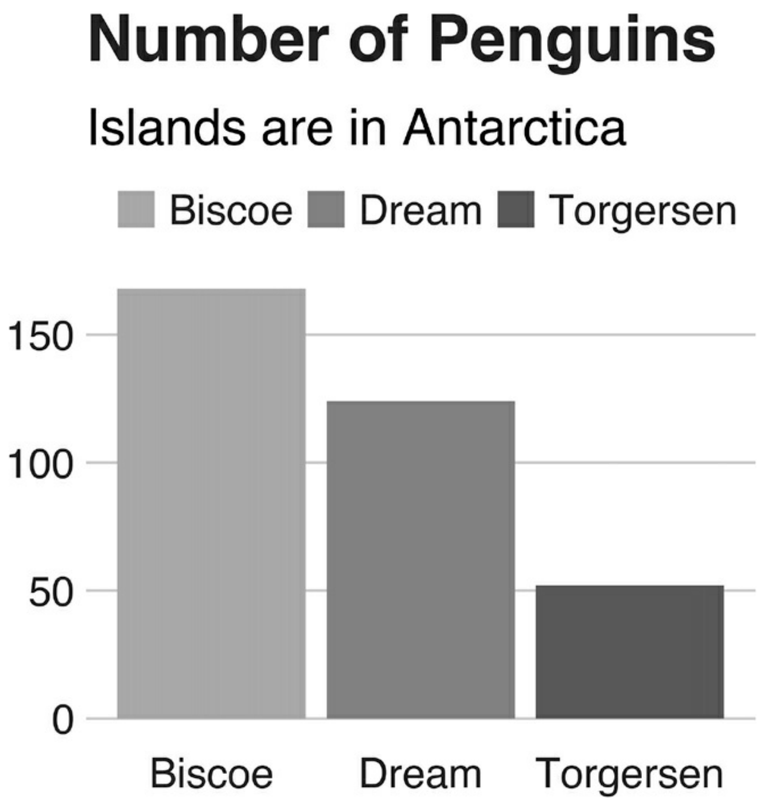


Figure 3-2: The same chart with BBC style applied

Vastly different, right? The font size is larger, the legend is on top, there are no axis titles, the grid lines are stripped down, and the background is white. Let's look at these changes one by one.

The BBC Theme Components

You've just seen the difference that the `bbc_style()` function makes to a basic chart. This section walks you through the function's code, with some minor tweaks for readability. Functions are discussed further in Chapter 12.

Function Definition

The first line gives the function a name and indicates that what follows is, in fact, a function definition:

```
bbc_style <- function() {
  font <- "Helvetica"

  ggplot2::theme(
--snip--
```

The code then defines a variable called `font` and assigns it the value `Helvetica`. This allows later sections to simply use `font` rather than repeating `Helvetica` multiple times. If the BBC team ever wanted to use a different font, they could change `Helvetica` here to, say, `Comic Sans` and it would update the font for all of the BBC plots (though I suspect higher-ups at the BBC might not be on board with that choice).

Historically, working with custom fonts in R was notoriously tricky, but recent changes have made the process much simpler. To ensure that custom fonts such as Helvetica work in ggplot, first install the `systemfonts` and `ragg` packages by running this code in the console:

```
install.packages(c("systemfonts", "ragg"))
```

The `systemfonts` package allows R to directly access fonts you've installed on your computer, and `ragg` allows ggplot to use those fonts when generating plots.

Next, select **Tools ▸ Global Options** from RStudio's main menu bar. Click the **Graphics** menu at the top of the interface and, under the Backend option, select **AGG**. This change should ensure that RStudio renders the previews of any plots with the `ragg` package. With these changes in place, you should be able to use any fonts you'd like (assuming you have them installed) in the same way that the `bbc_style()` function uses Helvetica.

After specifying the font to use, the code calls ggplot's `theme()` function. Rather than first loading ggplot with `library(ggplot2)` and then calling its `theme()` function, the `ggplot2::theme()` syntax indicates in one step that the `theme()` function comes from the `ggplot2` package. You'll write code in this way when making an R package in Chapter 12.

Nearly all of the code in `bbc_style()` exists within this `theme()` function. Remember from Chapter 2 that `theme()` makes additional tweaks to an existing theme; it isn't a complete theme like `theme_light()`, which will change the whole look and feel of your plot. In other words, by jumping straight into the `theme()` function, `bbc_style()` makes adjustments to the ggplot defaults. As you'll see, the `bbc_style()` function does a lot of tweaking.

Text

The first code section within the `theme()` function formats the text:

```
    plot.title = ggplot2::element_text(
      family = font,
      size = 28,
      face = "bold",
      color = "#222222"
    ),
    plot.subtitle = ggplot2::element_text(
      family = font,
      size = 22,
      margin = ggplot2::margin(9, 0, 9, 0)
    ),
    plot.caption = ggplot2::element_blank(),
--snip--
```

To make changes to the title, subtitle, and caption, it follows this pattern:

```
area_of_chart = element_type(
  property = value
)
```

For each area, this code specifies the element type: `element_text()`, `element_line()`, `element_rect()`, or `element_blank()`. Within the element type is where you assign values to properties—for example, setting the font family (the property) to Helvetica (the value). The `bbc_style()` function uses the various `element_` functions to make tweaks, as you'll see later in this chapter.

NOTE

For additional ways to customize pieces of your plots, see the ggplot2 package documentation (https://ggplot2.tidyverse.org/reference/element.xhtml), which provides a comprehensive list.

One of the main adjustments the `bbc_style()` function makes is bumping up the font size to help with legibility, especially when plots made with the `bbplot` package are viewed on smaller mobile devices. The code first formats the title (with `plot.title`) using Helvetica 28-point bold font in a nearly black color (the hex code #222222). The subtitle (`plot.subtitle`) is 22-point Helvetica.

The `bbc_style()` code also adds some spacing between the title and subtitle with the `margin()` function, specifying the value in points for the top (`9`), right (`0`), bottom (`9`), and left (`0`) sides. Finally, the `element_blank()` function removes the default caption (set through the `caption` argument in the `labs()` function), "Data from palmer penguins package." (As mentioned earlier, the `finalise_plot()` function in the `bbplot` package adds elements, including an updated caption and the BBC logo, to the bottom of the plots.) Figure 3-3 shows these changes.

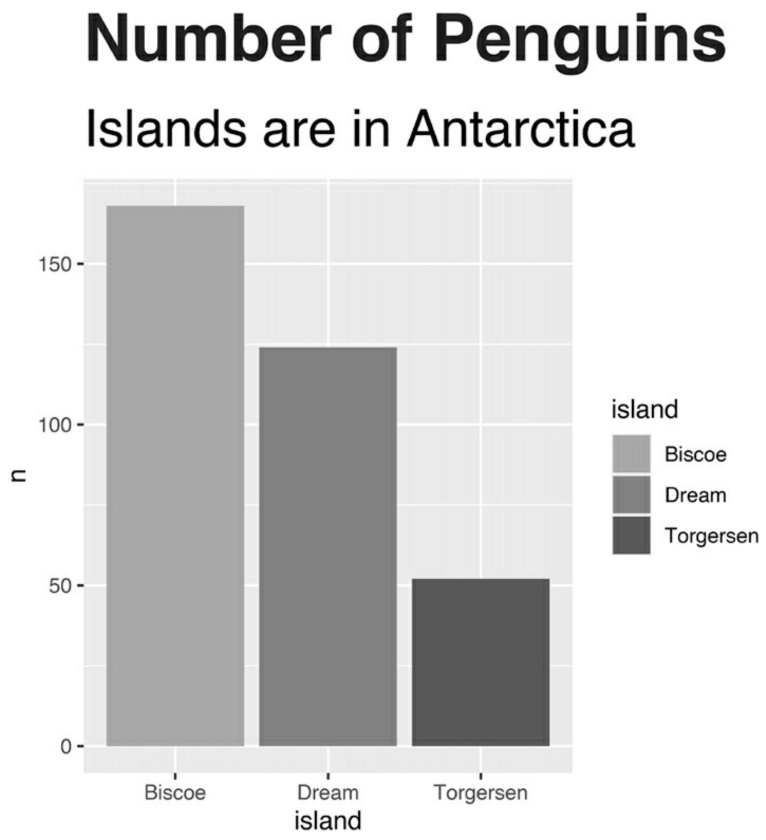


Figure 3-3: The penguin chart with the text and margin formatting changes

With these changes in place, you’re on your way to the BBC look.

Legend

Next up is formatting the legend, positioning it above the plot and left-aligning its text:

```
legend.position = "top",
legend.text.align = 0,
legend.background = ggplot2::element_blank(),
legend.title = ggplot2::element_blank(),
legend.key = ggplot2::element_blank(),
legend.text = ggplot2::element_text(
  family = font,
  size = 18,
  color = "#222222"
),
```

This code removes the legend background (which would show up only if the background color of the entire plot weren't white), the title, and the legend key (the borders on the boxes that show the island names, just barely visible in Figure 3-3). Finally, the code sets the legend's text to 18-point Helvetica with the same nearly black color. Figure 3-4 shows the result.

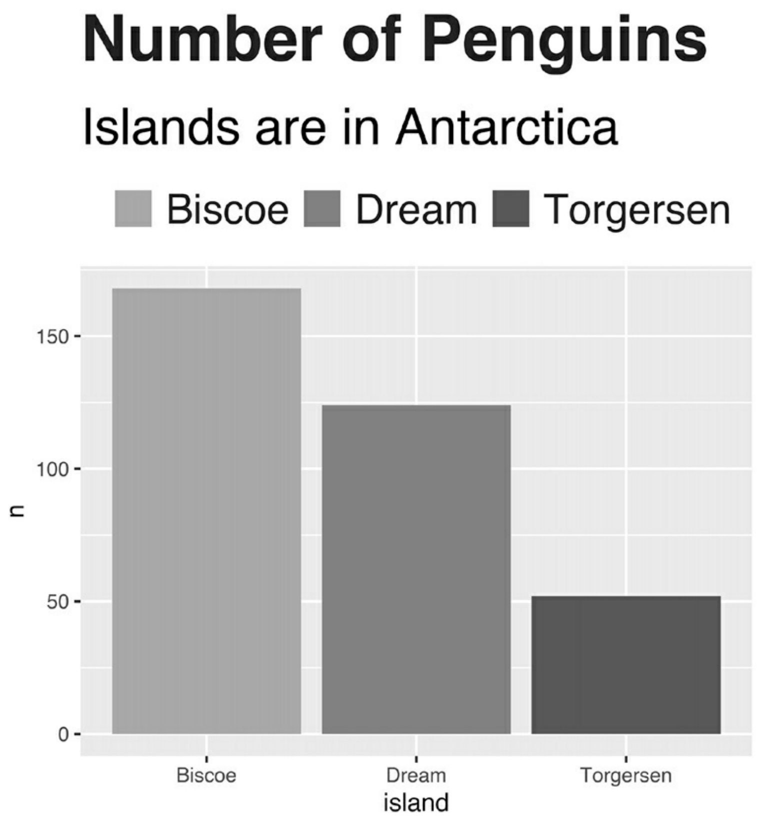


Figure 3-4: The penguin chart with changes to the legend

The legend is looking better, but now it's time to format the rest of the chart so it matches.

Axes

The code first removes the axis titles because they tend to take up a lot of chart real estate, and you can use the title and subtitle to clarify what the axes show:

```
axis.title = ggplot2::element_blank(),
axis.text = ggplot2::element_text(
  family = font,
  size = 18,
        color = "#222222"
      ),
      axis.text.x = ggplot2::element_text(margin = ggplot2::mar
gin(5, b = 10)),
      axis.ticks = ggplot2::element_blank(),
      axis.line = ggplot2::element_blank(),
```

All text on the axes becomes 18-point Helvetica and nearly black. The text on the x-axis (Biscoe, Dream, and Torgersen) gets a bit of spacing around it. Finally, both axes’ ticks and lines are removed. Figure 3-5 shows these changes, although the removal of the axis lines doesn’t make a difference to the display here.

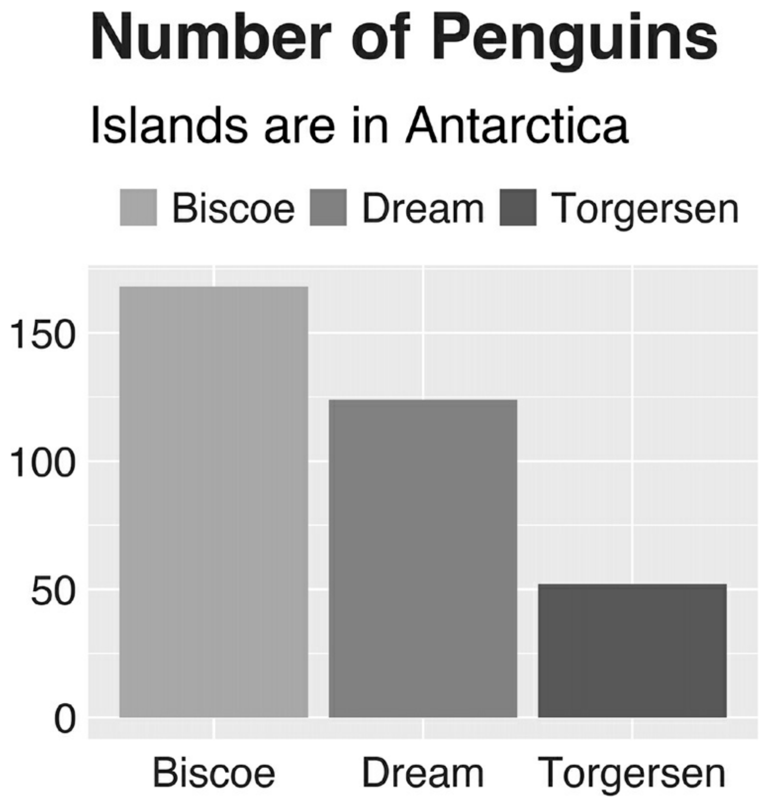


Figure 3-5: The penguin chart with axis formatting changes

The axis text matches the legend text, and the axis tick marks and lines are gone.

Grid Lines

Now for the grid lines:

```
    panel.grid.minor = ggplot2::element_blank(),
    panel.grid.major.y = ggplot2::element_line(color = "#cbcb
cb"),
    panel.grid.major.x = ggplot2::element_blank(),
--snip--
```

The approach here is fairly straightforward: this code removes minor grid lines for both axes, removes major grid lines on the x-axis, and keeps major grid lines on the y-axis but makes them a light gray (the `#cbcbcb` hex code). Figure 3-6 shows the result.

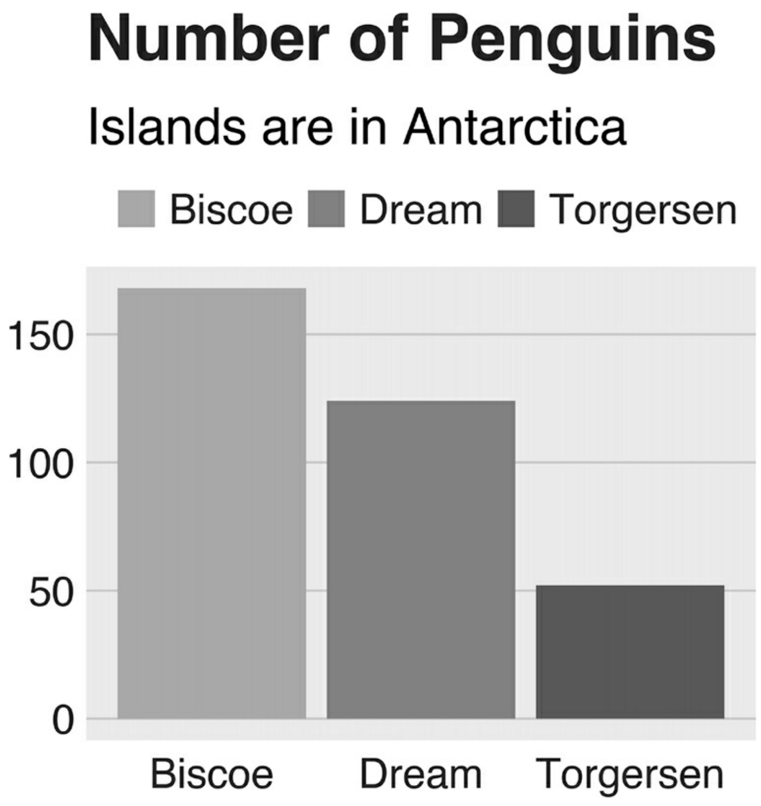


Figure 3-6: The penguin chart with adjustments to the grid lines

Notice that the grid lines on the x-axis have disappeared.

Background

The previous iteration of the plot still has a gray background. The `bbc_style()` function removes it with the following code:

```
    panel.background = ggplot2::element_blank(),
```

Figure 3-7 shows the resulting plot.

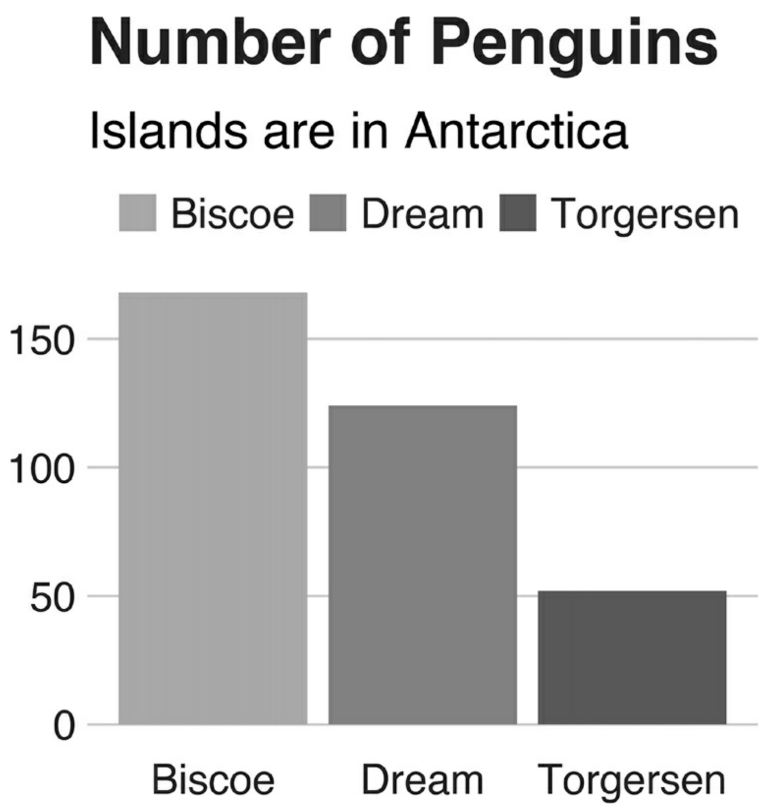


Figure 3-7: The chart with the gray background removed

You've nearly re-created the penguin plot using the `bbc_style()` function.

Small Multiples

The `bbc_style()` function contains a bit more code to modify `strip.background` and `strip.text`. In ggplot, the *strip* refers to the text above faceted charts like the ones discussed in Chapter 2. Next, you'll turn your penguin chart into a faceted chart to see these components of the BBC's theme. I've used the code from the `bbc_style()` function, minus the sections that deal with small multiples, to make Figure 3-8.

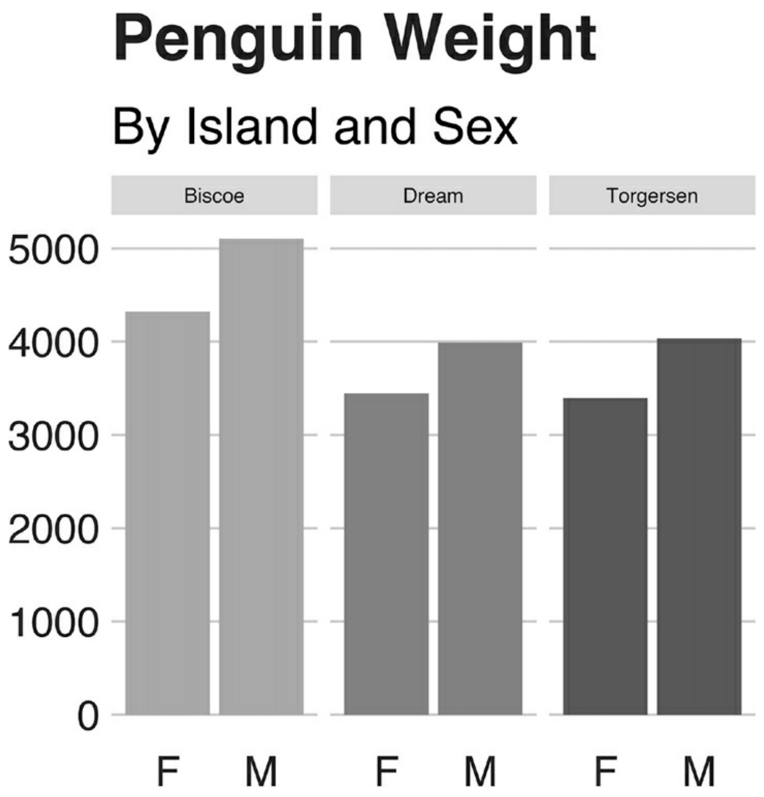


Figure 3-8: The faceted chart with no changes to the strip text formatting

Using the `facet_wrap()` function to make a small multiples chart leaves you with one chart per island, but by default, the text above each small multiple is noticeably smaller than the rest of the chart. What's more, the gray background behind the text stands out because you've already removed the gray background from the other parts of the chart. The consistency you've worked toward is now compromised, with small text that is out of proportion to the other chart text and a gray background that sticks out like a sore thumb.

The following code changes the strip text above each small multiple:

```
    strip.background = ggplot2::element_rect(fill = "white"),
    strip.text = ggplot2::element_text(size = 22, hjust = 0)
  )
  --snip--
```

This code removes the background (or, more accurately, colors it white). Then it makes the text larger, bold, and left-aligned using `hjust = 0`. Note that I did have to make the text size slightly smaller than in the actual chart to fit the book, and I added code to make it bold. Figure 3-9 shows the result.

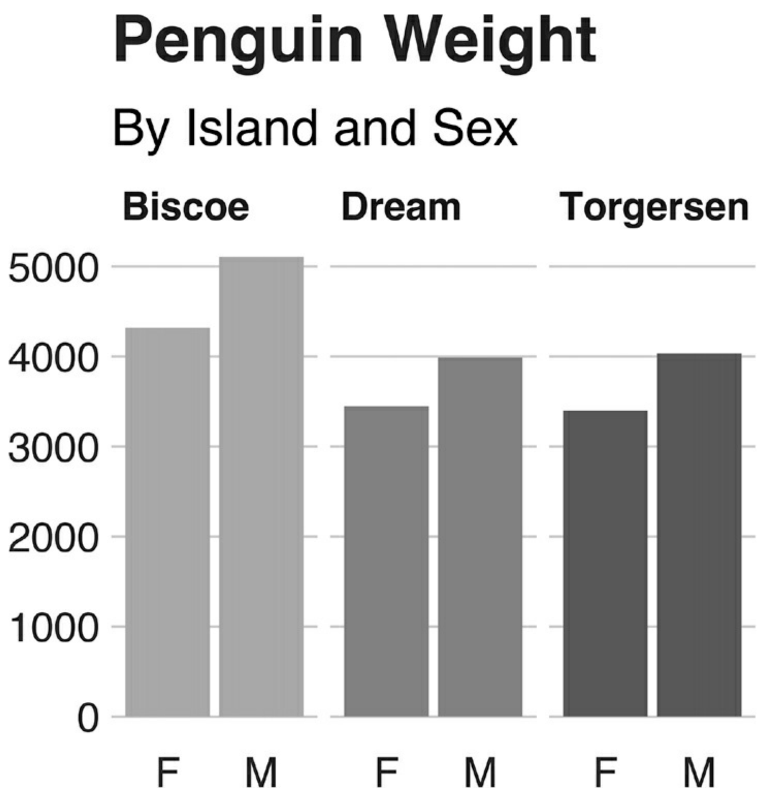


Figure 3-9: The small multiples chart in the BBC style

If you look at any chart on the BBC website, you'll see how similar it looks to your own. The tweaks in the `bbc_style()` function to the text formatting, legends, axes, grid lines, and backgrounds show up in charts viewed by millions of people worldwide.

Color

You might be thinking, *Wait, what about the color of the bars? Doesn't the theme change those?* This is a common point of confusion, but the answer is that it doesn't. The documentation for the `theme()` function explains why this is the case: "Themes are a powerful way to customize the non-data components of your plots: i.e. titles, labels, fonts, background, gridlines, and legends." In other words, ggplot themes change the elements of the chart that aren't mapped to data.

Plots, on the other hand, use color to communicate information about data. In the faceted chart, for instance, the `fill` property is mapped to the island (Biscoe is salmon, Dream is green, and Torgersen is blue). As you saw in Chapter 2, you can change the fill using the various `scale_fill_` functions. In the world of ggplot, these `scale_` functions control color, while the custom themes control the chart's overall look and feel.

Summary

When Stylianou and Guibourg started developing a custom theme for the BBC, they had one question: Would they be able to create graphs in R that could go directly onto the BBC website? Using ggplot, they succeeded. The `bbplot` package allowed them to make plots with a consistent look and feel that followed BBC standards and, most important, did not require a designer's help.

You can see many of the principles of high-quality data visualization discussed in Chapter 2 in this custom theme. In particular, the removal of extraneous elements (axis titles and grid lines, for instance) helps keep the focus on the data itself. And because applying the theme requires users to add only a single line to their ggplot code, it was easy to get others on board. They had only to append `bbc_style()` to their code to produce a BBC-style plot.

Over time, others at the BBC noticed the data journalism team's production-ready graphs and wanted to make their own. The team members set up R trainings for their colleagues and developed a "cookbook" (*https://bbc.github.io/rcookbook/*) showing how to make various types of charts. Soon, the quality and quantity of BBC's data visualization exploded. Stylianou told me, "I don't think there's been a day where someone at the BBC hasn't used the package to produce a graphic."

Now that you've seen how custom ggplot themes work, try making one of your own. After all, once you've written the code, you can apply it with only one line of code.

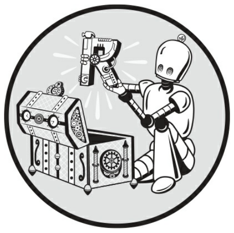
Additional Resources

- BBC Visual and Data Journalism Team, "BBC Visual and Data Journalism Cookbook for R Graphics," GitHub, January 24, 2019, *https://bbc.github.io/rcookbook/*.

- BBC Visual and Data Journalism Team, “How the BBC Visual and Data Journalism Team Works with Graphics in R,” Medium, February 1, 2019, *https://medium.com/bbc-visual-and-data-journalism/how-the-bbc-visual-and-data-journalism-team-works-with-graphics-in-r-ed0b35693535*.

4

MAPS AND GEOSPATIAL DATA



When I first started learning R, I considered it a tool for working with numbers, not shapes, so I was surprised when I saw people using it to make maps. For example, developer Abdoul Madjid used R to make a map that visualizes rates of COVID-19 in the United States in 2021.

You might think you need specialized mapmaking software like ArcGIS to make maps, but it’s an expensive tool. And while Excel has added support for mapmaking in recent years, its features are limited (for example, you can’t use it to make maps based on street addresses). Even QGIS, an open source tool similar to ArcGIS, still requires learning new skills.

Using R for mapmaking is more flexible than using Excel, less expensive than using ArcGIS, and based on syntax you already know. It also lets you perform all of your data manipulation tasks with one tool and apply the principles of high-quality data visualization discussed in Chapter 2. In this chapter, you’ll work with simple features of geospatial data and examine Madjid’s code to understand how he created this map. You’ll also learn where to find geospatial data and how to use it to make your own maps.

A Brief Primer on Geospatial Data

You don’t need to be a GIS expert to make maps, but you do need to understand a few things about how geospatial data works, starting with its two main types: vector and raster. *Vector* data uses points, lines, and polygons to represent the world. *Raster* data, which often comes from digital photographs, ties each pixel in an image to a specific geographic location. Vector data tends to be easier to work with, and you'll be using it exclusively in this chapter.

In the past, working with geospatial data meant mastering competing standards, each of which required learning a different approach. Today, though, most people use the *simple features* model (often abbreviated as *sf*) for working with vector geospatial data, which is easier to understand. For example, to import simple features data about the state of Wyoming, enter the following:

```
library(sf)

wyoming <- read_sf("https://data.rfortherestofus.com/wyoming.
geojson")
```

And then you can look at the data like so:

```
> wyoming
#> Simple feature collection with 1 feature and 1 field
#> Geometry type: POLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -111.0546 ymin: 40.99477 xmax: -104.0
522 ymax: 45.00582
#> Geodetic CRS:  WGS 84
#>   NAME    geometry
#> 1 Wyoming POLYGON ((-111.0449 43.3157...
```

The output has two columns, one for the state name (`NAME`) and another called `geometry`. This data looks like the data frames you've seen before, aside from two major differences.

First, there are five lines of metadata above the data frame. At the top is a line stating that the data contains one feature and one field. A *feature* is a row of data, and a *field* is any column containing nonspatial data. Second, the simple features data contains geographical data in a variable called `geometry`. Because the `geometry` column must be present for a data frame to be geospatial data, it isn’t counted as a field. Let’s look at each part of this simple features data.

The Geometry Type

The *geometry type* represents the shape of the geospatial data you’re working with and is typically shown in all caps. In this case, the relatively simple `POLYGON` type represents a single polygon. You can use ggplot to display this data by calling `geom_sf()`, a special geom designed to work with simple features data:

```
library(tidyverse)

wyoming %>%
  ggplot() +
  geom_sf()
```

Figure 4-1 shows the resulting map of Wyoming. It may not look like much, but I wasn’t the one who made Wyoming a nearly perfect rectangle!

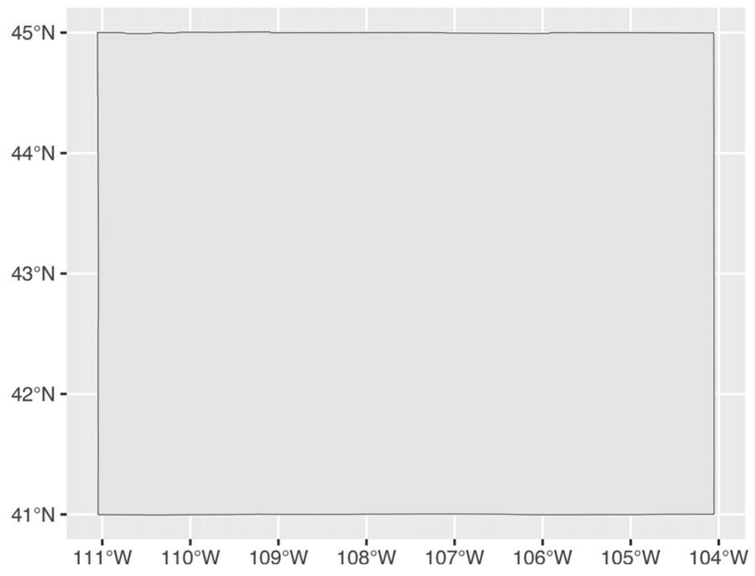


Figure 4-1: A map of Wyoming generated using POLYGON simple features data

Other geometry types used in simple features data include `POINT`, to display elements such as a pin on a map that represents a single location. For example, the map in Figure 4-2 uses `POINT` data to show a single electric vehicle charging station in Wyoming.

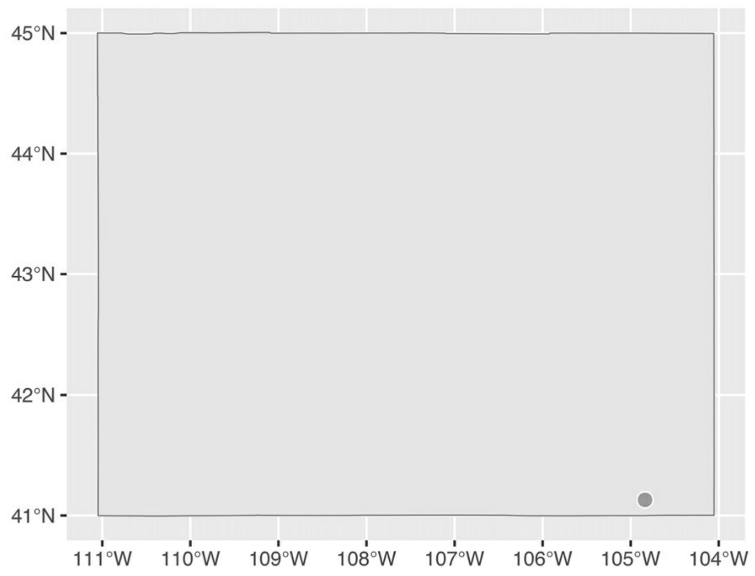


Figure 4-2: A map of Wyoming containing POINT simple features data

The `LINESTRING` geometry type is for a set of points that can be connected with lines and is often used to represent roads. Figure 4-3 shows a map that uses `LINESTRING` data to represent a section of US Highway 30 that runs through Wyoming.

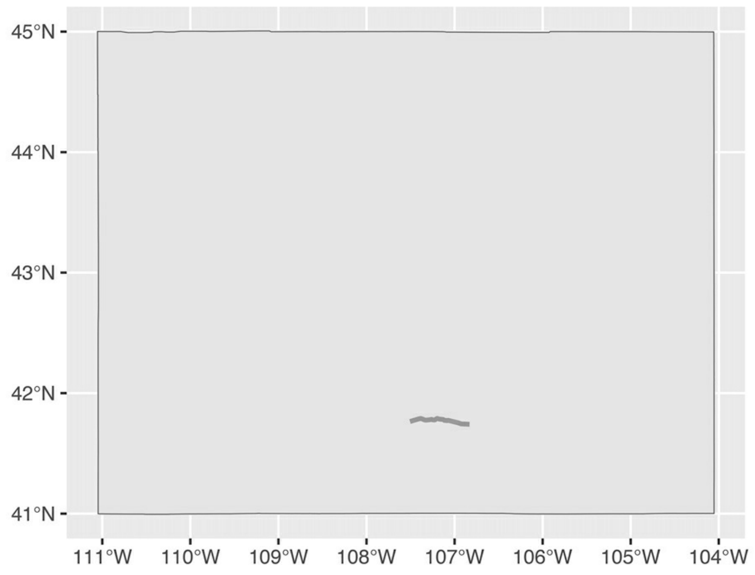


Figure 4-3: A road represented using LINESTRING simple features data

Each of these geometry types has a `MULTI` variation (`MULTIPOINT`, `MULTILINESTRING`, and `MULTIPOLYGON`) that combines multiple instances of the type in one row of data. For example, Figure 4-4 uses `MULTIPOINT` data to show all electric vehicle charging stations in Wyoming.

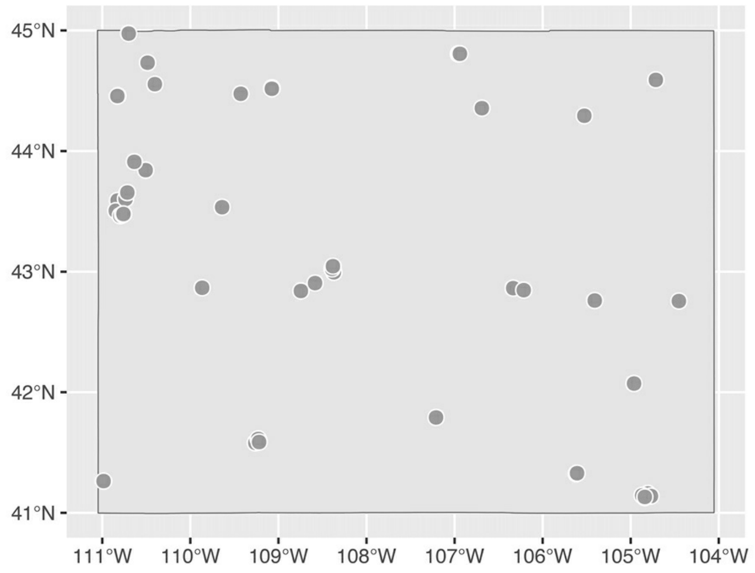


Figure 4-4: Using MULTIPOINT data to represent multiple electric vehicle charging stations

Likewise, you can use `MULTILINESTRING` data to show not just one road but all major roads in Wyoming (Figure 4-5).

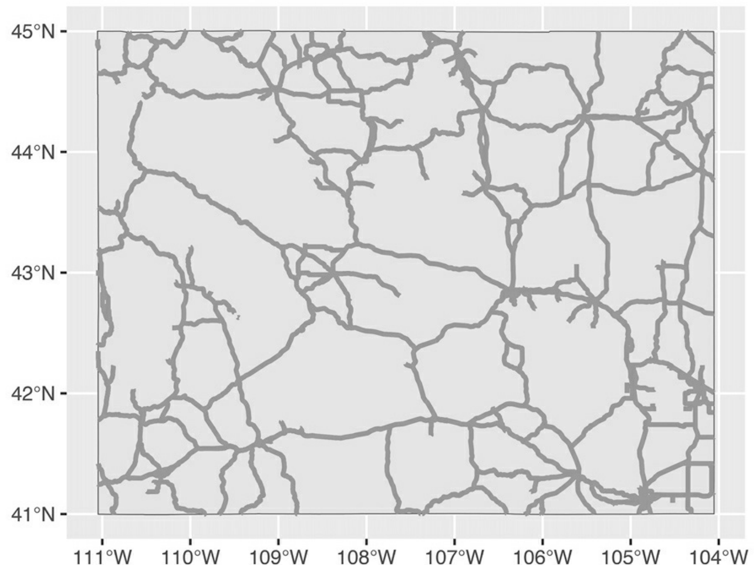


Figure 4-5: Using MULTILINESTRING data to represent several roads

Finally, you could use `MULTIPOLYGON` data, for example, to depict a state made up of multiple polygons. The following data represents the 23 counties in the state of Wyoming:

```
#> Simple feature collection with 23 features and 1 field
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -111.0546 ymin: 40.99477 xmax: -104.0
522 ymax: 45.00582
#> Geodetic CRS:  WGS 84
#> First 10 features:
#>          NAME        geometry
#> 34       Lincoln     MULTIPOLYGON (((-111.0472 4...
#> 104      Fremont     MULTIPOLYGON (((-109.4582 4...
#> 121      Uinta       MULTIPOLYGON (((-110.6068 4...
#> 527      Big Horn    MULTIPOLYGON (((-108.5923 4...
#> 551      Hot Springs MULTIPOLYGON (((-109.1714 4...
#> 601      Washakie    MULTIPOLYGON (((-107.6335 4...
#> 769      Converse    MULTIPOLYGON (((-105.6985 4...
#> 970      Sweetwater  MULTIPOLYGON (((-110.0489 4...
#> 977      Crook       MULTIPOLYGON (((-105.0856 4...
#> 1097     Carbon      MULTIPOLYGON (((-106.9129 4...
```

As you can see on the second line, the geometry type of this data is `MULTIPOLYGON`. In addition, the repeated `MULTIPOLYGON` text in the `geometry` column indicates that each row contains a shape of type `MULTIPOLYGON`. Figure 4-6 shows a map made with this data.

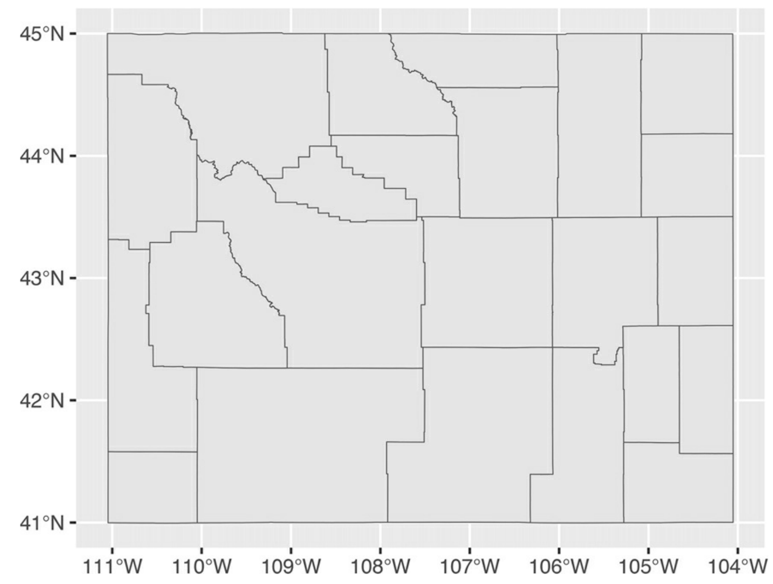


Figure 4-6: A map of Wyoming counties

Notice that the map is made up entirely of polygons.

The Dimensions

Next, the geospatial data frame contains the data's *dimensions*, or the type of geospatial data you're working with. In the Wyoming example, it looks like `Dimension: XY`, meaning the data is two-dimensional, as in the case of all the geospatial data used in this chapter. There are two other dimensions (`Z` and `M`) that you'll see much more rarely. I'll leave them for you to investigate further.

The Bounding Box

The penultimate element in the metadata is the *bounding box*, which represents the smallest area in which you can fit all of your geospatial data. For the `wyoming` object, it looks like this:

```
Bounding box:  xmin: -111.0569 ymin: 40.99475 xmax: -104.0522
 ymax: 45.0059
```

The `ymin` value of `40.99475` and `ymax` value of `45.0059` represent the lowest and highest latitudes, respectively, that the state's polygon can fit into. The x-values do the same for the longitude. Bounding boxes are calculated automatically, and typically you don’t have to worry about altering them.

The Coordinate Reference System

The last piece of metadata specifies the *coordinate reference system* used to project the data when it’s plotted. The challenge with representing any geospatial data is that you’re displaying information about the three-dimensional Earth on a two-dimensional map. Doing so requires choosing a coordinate reference system that determines what type of correspondence, or *projection*, to use when making the map.

The data for the Wyoming counties map includes the line `Geodetic CRS: WGS 84`, indicating the use of a coordinate reference system known as *WGS84*. To see a different projection, check out the same map using the *Albers equal-area conic convenience projection*. While Wyoming looked perfectly horizontal in Figure 4-6, the version in Figure 4-7 appears to be tilted.

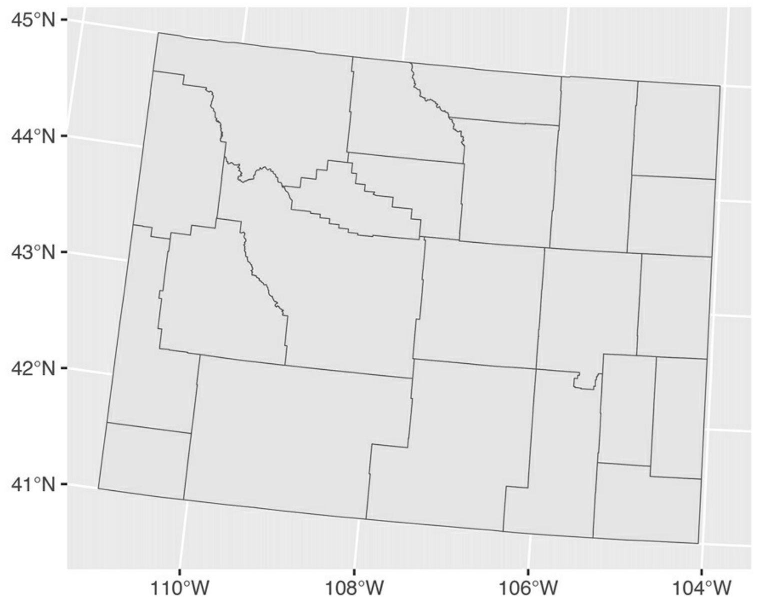


Figure 4-7: A map of Wyoming counties using the Albers equal-area conic convenience projection

If you’re wondering how to change projections when making maps of your own, fear not: you’ll see how to do this when we look at Madjid’s map in the next section. And if you want to know how to choose appropriate projections for your maps, check out “Using Appropriate Projections” on page 81.

The geometry Column

In addition to the metadata, simple features data differs from traditional data frames in another respect: its `geometry` column. As you might have guessed from the name, this column holds the data needed to draw the maps.

To understand how this works, consider the connect-the-dots drawings you probably completed as a kid. As you added lines to connect one point to the next, the subject of your drawing became clearer. The `geometry` column is similar. It has a set of numbers, each of which corresponds to a point. If you’re using `LINESTRING`/`MULTILINESTRING` or `POLYGON`/`MULTIPOLYGON` simple features data, ggplot uses the numbers in the `geometry` column to draw each point and then adds lines to connect the points. If you’re using `POINT`/`MULTIPOINT` data, it draws the points but doesn’t connect them.

Once again, thanks to R, you never have to worry about these details or look in any depth at the `geometry` column.

Re-creating the COVID-19 Map

Now that you understand the basics of geospatial data, let’s walk through the code Madjid used to make his COVID-19 map. Shown in Figure 4-8, it makes use of the geometry types, dimensions, bounding boxes, projections, and the `geometry` column just discussed.

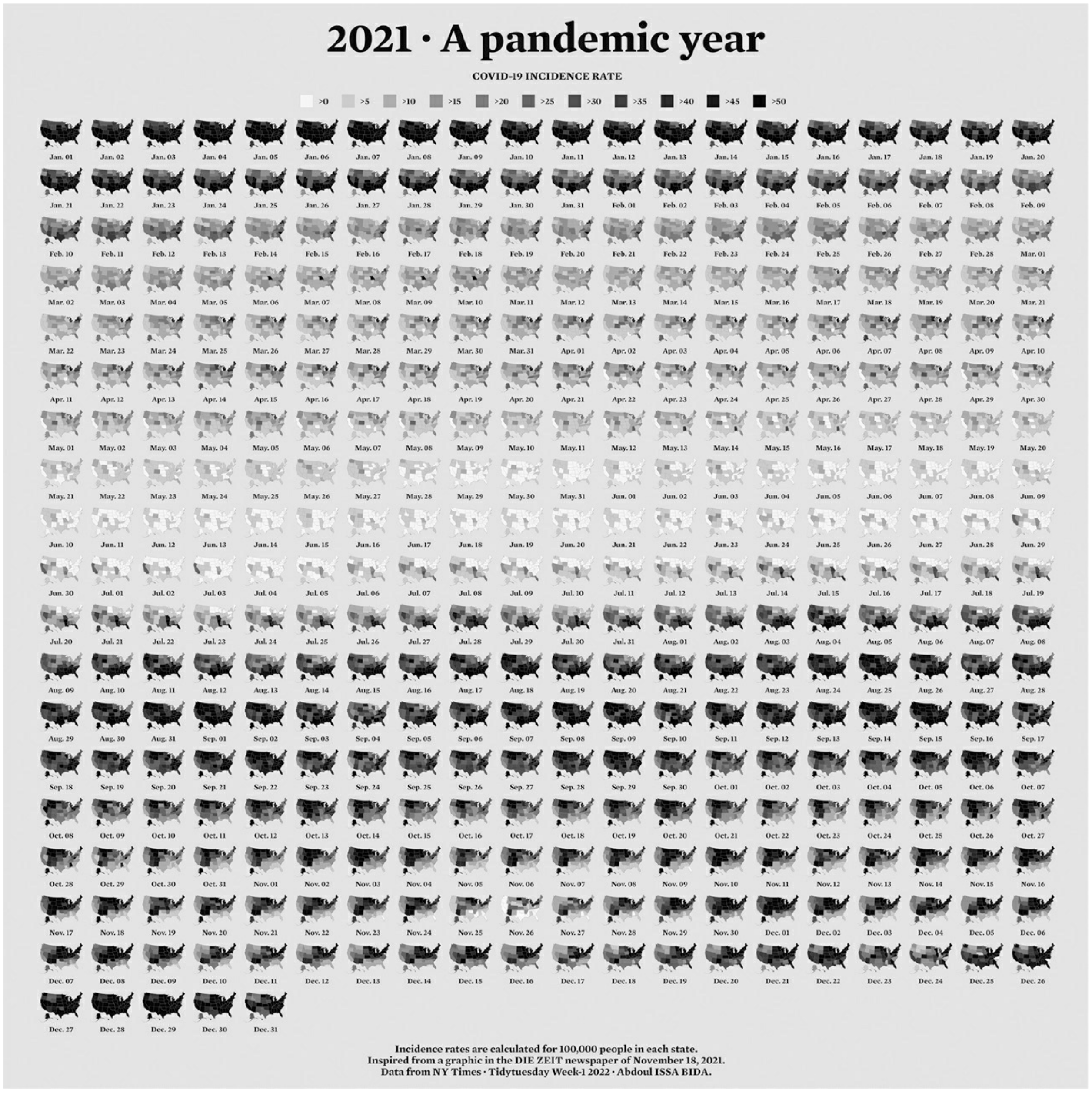

2021 · A pandemic year
COVID-19 INCIDENCE RATE
>0
>5
>10
>15
>20
>25
>30
>35
>40
>45
>50
Incidence rates are calculated for 100,000 people in each state.
Inspired from a graphic in the DIE ZEIT newspaper of November 18, 2021.
Data from NY Times · Tidytuesday Week-1 2022 · Abdoul ISSA BIDA.

Figure 4-8: Abdoul Madjid's map of COVID-19 in the United States in 2021

I've made some small modifications to the code to make the final map fit on the page. You'll begin by loading a few packages:

```
library(tidyverse)
library(albersusa)
library(sf)
library(zoo)
library(colorspace)
```

The `albersusa` package will give you access to geospatial data. Install it as follows:

```
remotes::install_github("hrbrmstr/albersusa")
```

You can install all of the other packages using the standard `install.packages()` code. You'll use the `tidyverse` to import data, manipulate it, and plot it with ggplot. The `sf` package will enable you to change the coordinate reference system and use an appropriate projection for the data. The `zoo` package has functions for calculating rolling averages, and the `colorspace` package gives you a color scale that highlights the data well.

Importing the Data

Next, you'll import the data you need: COVID-19 rates by state over time, state populations, and geospatial information. Madjid imported each of these pieces of data separately and then merged them, and you'll do the same.

The COVID-19 data comes directly from the *New York Times*, which publishes daily case rates by state as a CSV file on its GitHub account. To import it, enter the following:

```
covid_data <- read_csv("https://data.rfortherestofus.com/covi
d-us-states.csv") %>%
  select(-fips)
```

Federal Information Processing Standards (FIPS) are numeric codes used to represent states, but you'll reference states by their names instead, so the line `select(-fips)` drops the `fips` variable.

Looking at this data, you can see the arrival of the first COVID-19 cases in the United States in January 2020:

```
#> # A tibble: 56,006 × 4
#>    date       state      cases deaths
#>    <date>     <chr>      <dbl>  <dbl>
#>  1 2020-01-21 Washington     1      0
#>  2 2020-01-22 Washington     1      0
#>  3 2020-01-23 Washington     1      0
#>  4 2020-01-24 Illinois       1      0
#>  5 2020-01-24 Washington     1      0
#>  6 2020-01-25 California     1      0
#>  7 2020-01-25 Illinois       1      0
#>  8 2020-01-25 Washington     1      0
#>  9 2020-01-26 Arizona        1      0
#> 10 2020-01-26 California     2      0
--snip--
```

Madjid’s map shows per capita rates (the rates per 100,000 people) rather than absolute rates (the rates without consideration for a state’s population). So, to re-create his maps, you also need to obtain data on each state’s population. Download this data as a CSV as follows:

```
usa_states <- read_csv("https://data.rfortherestofus.com/popu
lation-by-state.csv") %>%
  select(State, Pop)
```

This code imports the data, keeps the `State` and `Pop` (population) variables, and saves the data as an object called `usa_states`. Here’s what `usa_states` looks like:

```
#> # A tibble: 52 × 2
#>    State          Pop
#>    <chr>          <dbl>
#>  1 California     39613493
#>  2 Texas          29730311
#>  3 Florida        21944577
#>  4 New York       19299981
#>  5 Pennsylvania   12804123
#>  6 Illinois       12569321
#>  7 Ohio           11714618
#>  8 Georgia        10830007
#>  9 North Carolina 10701022
#> 10 Michigan        9992427
--snip--
```

Finally, import the geospatial data and save it as an object called `usa_states_geom` like so:

```
usa_states_geom <- usa_sf() %>%
  select(name) %>%
  st_transform(us_laea_proj)
```

The `usa_sf()` function from the `albersusa` package gives you simple features data for all US states. Conveniently, it places Alaska and Hawaii at a position and scale that make them easy to see. This data includes multiple variables, but because you need only the state names, this code keeps just the `name` variable.

The `st_transform()` function from the `sf` package changes the coordinate reference system. The one used here comes from the `us_laea_proj` object in the `albersusa` package. This is the Albers equal-area conic convenience projection you used earlier to change the appearance of the Wyoming counties map.

Calculating Daily COVID-19 Cases

The `covid_data` data frame lists cumulative COVID-19 cases by state, but not the number of cases per day, so the next step is to calculate that number:

```
covid_cases <- covid_data %>%
  group_by(state) %>%
  mutate(
 ❶ pd_cases = lag(cases)
  ) %>%
❷ replace_na(list(pd_cases = 0)) %>%
  mutate(
 ❸ daily_cases = case_when(
      cases > pd_cases ~ cases - pd_cases,
      TRUE ~ 0
    )
) %>%
❹ ungroup() %>%
  arrange(state, date)
```

The `group_by()` function calculates totals for each state, then creates a new variable called `pd_cases`, which represents the number of cases in the previous day (the `lag()` function is used to assign data to this variable) ❶. Some days don't have case counts for the previous day, so set this value to `0` using the `replace_na()` function ❷.

Next, this code creates a new variable called `daily_cases` ❸. To set the value of this variable, use the `case_when()` function to create a condition: if the `cases` variable (which holds the cases on that day) is greater than the `pd_cases` variable (which holds cases from one day prior), then `daily_cases` is equal to `cases` minus `pd_cases`. Otherwise, you set `daily_cases` to be equal to `0`.

Finally, because you grouped the data by state at the beginning of the code, now you need to remove this grouping using the `ungroup()` function before arranging the data by state and date ❹.

Here's the resulting `covid_cases` data frame:

```
#> # A tibble: 56,006 × 6
#>    date       state   cases deaths pd_cases daily_cases
#>    <date>     <chr>   <dbl>  <dbl>    <dbl>       <dbl>
#>  1 2020-03-13 Alabama     6      0        0           6
#>  2 2020-03-14 Alabama    12      0        6           6
#>  3 2020-03-15 Alabama    23      0       12          11
#>  4 2020-03-16 Alabama    29      0       23           6
#>  5 2020-03-17 Alabama    39      0       29          10
#>  6 2020-03-18 Alabama    51      0       39          12
#>  7 2020-03-19 Alabama    78      0       51          27
#>  8 2020-03-20 Alabama   106      0       78          28
#>  9 2020-03-21 Alabama   131      0      106          25
#> 10 2020-03-22 Alabama   157      0      131          26
--snip--
```

In the next step, you'll make use of the new `daily_cases` variable.

Calculating Incidence Rates

You're not quite done calculating values. The data that Madjid used to make his map didn't include daily case counts. Instead, it contained a five-day rolling average of cases per 100,000 people. A *rolling average* is the average case rate in a certain time period. Quirks of reporting (for example, not reporting on weekends but instead rolling Saturday and Sunday cases into Monday) can make the value for any single day less reliable. Using a rolling average smooths out these quirks. Generate this data as follows:

```
covid_cases %>%
  mutate(roll_cases = rollmean(
    daily_cases,
    k = 5,
    fill = NA
  ))
```

This code creates a new data frame called `covid_cases_rm` (where *rm* stands for rolling mean). The first step in its creation is to use the `rollmean()` function from the `zoo` package to create a `roll_cases` variable, which holds the average number of cases in the five-day period surrounding a single date. The `k` argument is the number of days for which you want to calculate the rolling average (`5`, in this case), and the `fill` argument determines what happens in cases like the first day, where you can't calculate a five-day rolling mean because there are no days prior to this day (Madjid set these values to `NA`).

After calculating `roll_cases`, you need to calculate per capita case rates. To do this, you need population data, so join the population data from the `usa_states` data frame with the `covid_cases` data like so:

```
covid_cases_rm <- covid_cases %>%
  mutate(roll_cases = rollmean(
    daily_cases,
    k = 5,
    fill = NA
    )
  ) %>%
  left_join(usa_states,
            by = c("state" = "State")) %>%
  drop_na(Pop)
```

To drop rows with missing population data, you call the `drop_na()` function with the `Pop` variable as an argument. In practice, this removes several US territories (American Samoa, Guam, the Northern Mariana Islands, and the Virgin Islands).

Next, you create a per capita case rate variable called `incidence_rate` by multiplying the `roll_cases` variable by 100,000 and then dividing it by the population of each state:

```
covid_cases_rm <- covid_cases_rm %>%
  mutate(incidence_rate = 10^5 * roll_cases / Pop) %>%
  mutate(
    incidence_rate = cut(
      incidence_rate,
      breaks = c(seq(0, 50, 5), Inf),
      include.lowest = TRUE
    ) %>%
      factor(labels = paste0(">", seq(0, 50, 5)))
  )
```

Rather than keeping raw values (for example, on June 29, 2021, Florida had a rate of 57.77737 cases per 100,000 people), you use the `cut()` function to convert the values into categories: values of `>0` (greater than zero), values of `>5` (greater than five), and values of `>50` (greater than 50).

The last step is to filter the data so it includes only 2021 data (the only year depicted in Madjid’s map) and then select just the variables (`state`, `date`, and `incidence_rate`) you’ll need to create the map:

```
covid_cases_rm %>%
  filter(date >= as.Date("2021-01-01")) %>%
  select(state, date, incidence_rate)
```

Here’s the final `covid_cases_rm` data frame:

```
#> # A tibble: 18,980 × 3
#>    state   date       incidence_rate
#>    <chr>   <date>     <fct>
#>  1 Alabama 2021-01-01 >50
#>  2 Alabama 2021-01-02 >50
#>  3 Alabama 2021-01-03 >50
#>  4 Alabama 2021-01-04 >50
#>  5 Alabama 2021-01-05 >50
#>  6 Alabama 2021-01-06 >50
#>  7 Alabama 2021-01-07 >50
#>  8 Alabama 2021-01-08 >50
#>  9 Alabama 2021-01-09 >50
#> 10 Alabama 2021-01-10 >50
--snip--
```

You now have a data frame that you can combine with your geospatial data.

Adding Geospatial Data

You've used two of the three data sources (COVID-19 case data and state population data) to create the `covid_cases_rm` data frame you'll need to make the map. Now it's time to use the third data source: the geospatial data you saved as `usa_states_geom`. Simple features data allows you to merge regular data frames and geospatial data (another point in its favor):

```
usa_states_geom %>%
  left_join(covid_cases_rm, by = c("name" = "state"))
```

This code merges the `covid_cases_rm` data frame into the geospatial data, matching the `name` variable from `usa_states_geom` to the `state` variable in `covid_cases_rm`.

Next, you create a new variable called `fancy_date` to format the date nicely (for example, Jan. 01 instead of 2021-01-01):

```
usa_states_geom_covid <- usa_states_geom %>%
  left_join(covid_cases_rm, by = c("name" = "state")) %>%
  mutate(fancy_date = fct_inorder(format(date, "%b. %d"))) %>
%
  relocate(fancy_date, .before = incidence_rate)
```

The `format()` function does the formatting, while the `fct_inorder()` function makes the `fancy_date` variable sort data by date (rather than, say, alphabetically, which would put August before January). Last, the `relocate()` function puts the `fancy_date` column next to the `date` column.

Save this data frame as `usa_states_geom_covid` and take a look at the result:

```
#> Simple feature collection with 18615 features and 4 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -2100000 ymin: -2500000 xmax: 2516374
 ymax: 732103.3
```

```
#> CRS:           +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_
0=0 +a=6370997
+b=6370997 +units=m +no_defs
#> First 10 features:
#>    name    date        fancy_date incidence_rate
#> 1  Arizona 2021-01-01  Jan. 01    >50
#> 2  Arizona 2021-01-02  Jan. 02    >50
#> 3  Arizona 2021-01-03  Jan. 03    >50
#> 4  Arizona 2021-01-04  Jan. 04    >50
#> 5  Arizona 2021-01-05  Jan. 05    >50
#> 6  Arizona 2021-01-06  Jan. 06    >50
#> 7  Arizona 2021-01-07  Jan. 07    >50
#> 8  Arizona 2021-01-08  Jan. 08    >50
#> 9  Arizona 2021-01-09  Jan. 09    >50
#> 10 Arizona 2021-01-10  Jan. 10    >50
#>    geometry
#> 1  MULTIPOLYGON (((-1111066 -8...
#> 2  MULTIPOLYGON (((-1111066 -8...
#> 3  MULTIPOLYGON (((-1111066 -8...
#> 4  MULTIPOLYGON (((-1111066 -8...
#> 5  MULTIPOLYGON (((-1111066 -8...
#> 6  MULTIPOLYGON (((-1111066 -8...
#> 7  MULTIPOLYGON (((-1111066 -8...
#> 8  MULTIPOLYGON (((-1111066 -8...
#> 9  MULTIPOLYGON (((-1111066 -8...
#> 10 MULTIPOLYGON (((-1111066 -8...
```

You can see the metadata and `geometry` columns discussed earlier in the chapter.

Making the Map

It took a lot of work to end up with the surprisingly simple `usa_states_geom_covid` data frame. While the data may be simple, the code Madjid used to make his map is quite complex. This section walks you through it in pieces.

The final map is actually multiple maps, one for each day in 2021. Combining 365 days makes for a large final product, so instead of showing the code for every single day, filter the `usa_states_geom_covid` to show just the first six days in January:

```
usa_states_geom_covid_six_days <- usa_states_geom_covid %>%
  filter(date <= as.Date("2021-01-06"))
```

Save the result as a data frame called `usa_states_geom_covid_six_days`. Here’s what this data looks like:

```
#> Simple feature collection with 306 features and 4 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -2100000 ymin: -2500000 xmax: 2516374
 ymax: 732103.3
#> CRS:           +proj=laea +lat_0=45 +lon_0=-100 +x_0=0 +y_
0=0 +a=6370997 +b=6370997 +unit...
#> First 10 features:
#>     name    date          fancy_date incidence_rate
#> 1   Arizona 2021-01-01    Jan. 01    >50
#> 2   Arizona 2021-01-02    Jan. 02    >50
#> 3   Arizona 2021-01-03    Jan. 03    >50
#> 4   Arizona 2021-01-04    Jan. 04    >50
#> 5   Arizona 2021-01-05    Jan. 05    >50
#> 6   Arizona 2021-01-06    Jan. 06    >50
#> 7  Arkansas 2021-01-01    Jan. 01    >50
#> 8  Arkansas 2021-01-02    Jan. 02    >50
#> 9  Arkansas 2021-01-03    Jan. 03    >50
#> 10 Arkansas 2021-01-04    Jan. 04    >50
#>    geometry
#> 1  MULTIPOLYGON (((-1111066 -8...
#> 2  MULTIPOLYGON (((-1111066 -8...
#> 3  MULTIPOLYGON (((-1111066 -8...
#> 4  MULTIPOLYGON (((-1111066 -8...
#> 5  MULTIPOLYGON (((-1111066 -8...
#> 6  MULTIPOLYGON (((-1111066 -8...
#> 7  MULTIPOLYGON (((557903.1 -1...
#> 8  MULTIPOLYGON (((557903.1 -1...
#> 9  MULTIPOLYGON (((557903.1 -1...
#> 10 MULTIPOLYGON (((557903.1 -1...
```

Madjid’s map is giant, as it includes all 365 days. The size of a few elements have been changed so that they fit in this book.

Generating the Basic Map

With your six days of data, you’re ready to make some maps. Madjid’s mapmaking code has two main parts: generating the basic map, then tweaking its appearance. First, you’ll revisit the three lines of code used to make the Wyoming maps, with some adornments to improve the quality of the visualization:

```
usa_states_geom_covid_six_days %>%
  ggplot() +
  geom_sf(
    aes(fill = incidence_rate),
    size = .05,
    color = "grey55"
  ) +
  facet_wrap(
    vars(fancy_date),
    strip.position = "bottom"
  )
```

The `geom_sf()` function plots the geospatial data, modifying a couple of arguments: `size = .05` makes the state borders less prominent and `color = "grey55"` sets them to a medium-gray color. Then, the `facet_wrap()` function is used for the faceting (that is, to make one map for each day). The `vars(fancy_date)` code specifies that the `fancy_date` variable should be used for the faceted maps, and `strip.position = "bottom"` moves the labels Jan. 01, Jan. 02, and so on to the bottom of the maps. Figure 4-9 shows the result.

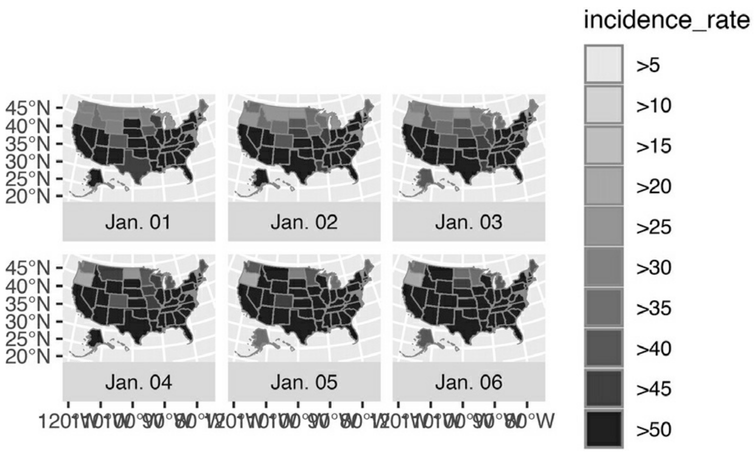


Figure 4-9: A map showing the incidence rate of COVID-19 for the first six days of 2021

Having generated the basic map, now you’ll make it look good.

Applying Data Visualization Principles

From now on, all of the code that Madjid uses is to improve the appearance of the maps. Many of the tweaks shown here should be familiar if you’ve read Chapter 2, highlighting a benefit of making maps with ggplot: you can apply the same data visualization principles you learned about when making charts.

```
usa_states_geom_covid_six_days %>%
  ggplot() +
  geom_sf(
    aes(fill = incidence_rate),
    size = .05,
    color = "transparent"
  ) +
  facet_wrap(
    vars(fancy_date),
    strip.position = "bottom"
  ) +
  scale_fill_discrete_sequential(
    palette = "Rocket",
    name = "COVID-19 INCIDENCE RATE",
    guide = guide_legend(
      title.position = "top",
      title.hjust = .5,
      title.theme = element_text(
        family = "Times New Roman",
        size = rel(9),
        margin = margin(
          b = .1,
          unit = "cm"
        )
      ),
      nrow = 1,
      keyheight = unit(.3, "cm"),
      keywidth = unit(.3, "cm"),
      label.theme = element_text(
        family = "Times New Roman",
        size = rel(6),
        margin = margin(
          r = 5,
          unit = "pt"
        )
      )
    )
  ) +
  labs(
    title = "2021 · A pandemic year",
    caption = "Incidence rates are calculated for 100,000 peo
ple in each state.
                  Inspired from a graphic in the DIE ZEIT new
spaper of November 18, 2021.
                  Data from NY Times · Tidytuesday Week-1 202
2 · Abdoul ISSA BIDA."
  ) +
  theme_minimal() +
  theme(
    text = element_text(
      family = "Times New Roman",
      color = "#111111"
    ),
    plot.title = element_text(
      size = rel(2.5),
      face = "bold",
      hjust = 0.5,
      margin = margin(
        t = .25,
        b = .25,
        unit = "cm"
      )
    ),
    plot.caption = element_text(
      hjust = .5,
      face = "bold",
      margin = margin(
        t = .25,
        b = .25,
        unit = "cm"
      )
    ),
    strip.text = element_text(
      size = rel(0.75),
      face = "bold"
    ),
    legend.position = "top",
    legend.box.spacing = unit(.25, "cm"),
    panel.grid = element_blank(),
    axis.text = element_blank(),
    plot.margin = margin(
      t = .25,
      r = .25,
      b = .25,
      l = .25,
      unit = "cm"
    ),
    plot.background = element_rect(
      fill = "#e5e4e2",
      color = NA
    )
  )
```

The `scale_fill_discrete_sequential()` function, from the `colorspace` package, sets the color scale. This code uses the rocket palette (the same palette that Cédric Scherer and Georgios Karamanis used in Chapter 2) and changes the legend title to "COVID-19 INCIDENCE RATE." The `guide_legend()` function adjusts the position, alignment, and text properties of the title. The code then puts the colored squares in one row, adjusts their height and width, and tweaks the text properties of the labels (`>0`, `>5`, and so on).

Next, the `labs()` function adds a title and caption. Following `theme_minimal()`, the `theme()` function makes some design tweaks, including setting the font and text color; making the title and caption bold; and adjusting their size, alignment, and margins. The code then adjusts the size of the strip text (Jan. 01, Jan. 02, and so on) and makes it bold, puts the legend at the top of the maps, and adds a bit of spacing around it. Grid lines, as well as the longitude and latitude lines, are removed, and then the entire visualization gets a bit of padding and a light gray background.

There you have it! Figure 4-10 shows the re-creation of his COVID-19 map.

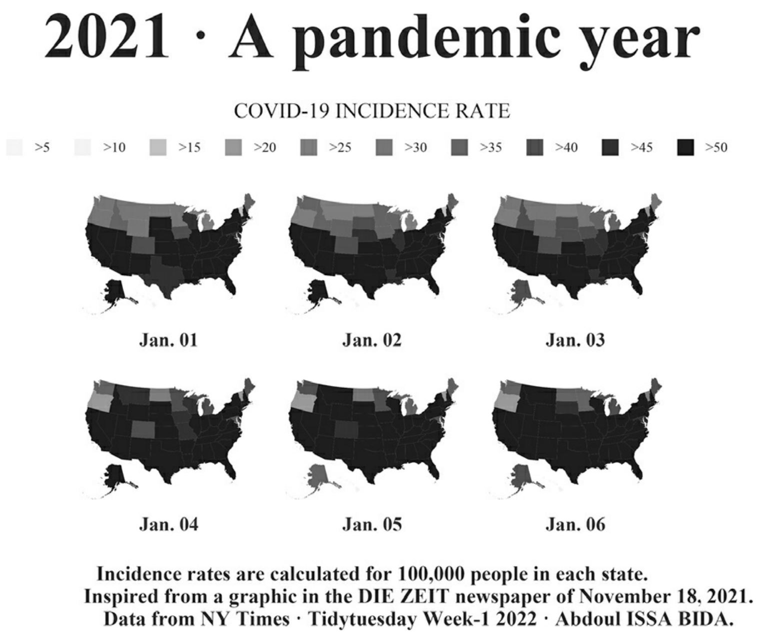

2021 · A pandemic year
COVID-19 INCIDENCE RATE
>5
>10
>15
>20
>25
>30
>35
>40
>45
>50
Jan. 01
Jan. 02
Jan. 03
Jan. 04
Jan. 05
Jan. 06
Incidence rates are calculated for 100,000 people in each state.
Inspired from a graphic in the DIE ZEIT newspaper of November 18, 2021.
Data from NY Times · Tidytuesday Week-1 2022 · Abdoul ISSA BIDA.

Figure 4-10: The re-creation of Abdoul Madjid's map

From data import and data cleaning to analysis and visualization, you've seen how Madjid made a beautiful map in R.

Making Your Own Maps

You may now be wondering, *Okay, great, but how do I actually make my own maps?* In this section you'll learn where you can find geospatial data, how to choose a projection, and how to prepare the data for mapping.

There are two ways to access simple features geospatial data. The first is to import raw data, and the second is to access it with R functions.

Importing Raw Data

Geospatial data can come in various formats. While ESRI shapefiles (with the *.shp* extension) are the most common, you might also encounter GeoJSON files (*.geojson*) like the ones we used in the Wyoming example at the beginning of this chapter, KML files (*.kml*), and others. Chapter 8 of *Geocomputation with R* by Robin Lovelace, Jakub Nowosad, and Jannes Muenchow discusses this range of formats.

The good news is that a single function can read pretty much any type of geospatial data: `read_sf()` from the `sf` package. Say you've downloaded geospatial data about US state boundaries from the website *geojson.xyz* in

GeoJSON format, then saved it in the *data* folder as *states.geojson*. To import this data, use the `read_sf()` function like so:

```
us_states <- read_sf(dsn = "https://data.rfortherestofus.com/
states.geojson")
```

The `dsn` argument (which stands for *data source name*) tells `read_sf()` where to find the file. You save the data as the object `us_states`.

Accessing Geospatial Data with R Functions

Sometimes you'll have to work with raw data in this way, but not always. That's because certain R packages provide functions for accessing geospatial data. Madjid used the `usa_sf()` function from the `albersusa` package to acquire his data. Another package for accessing geospatial data related to the United States, `tigris`, has a number of well-named functions for different types of data. For example, load the `tigris` package and run the `states()` function like so:

```
library(tigris)

states_tigris <- states(
  cb = TRUE,
  resolution = "20m",
  progress_bar = FALSE
)
```

The `cb = TRUE` argument opts out of using the most detailed shapefile and sets the resolution to a more manageable `20m` (1:20 million). Without these changes, the resulting shapefile would be large and slow to work with. Setting `progress_bar = FALSE` hides the messages that `tigris` generates as it loads data. The result is saved as `states_tigris`. The `tigris` package has functions to get geospatial data about counties, census tracts, roads, and more.

If you're looking for data outside the United States, the `rnaturalearth` package provides functions for importing geospatial data from across the world. For example, use `ne_countries()` to retrieve geospatial data about various countries:

```
library(rnaturalearth)

africa_countries <- ne_countries(
  returnclass = "sf",
  continent = "Africa"
)
```

This code uses two arguments: `returnclass = "sf"` to get data in simple features format, and `continent = "Africa"` to get only countries on the African continent. If you save the result to an object called `africa_countries`, you can plot the data on a map as follows:

```
africa_countries %>%
  ggplot() +
  geom_sf()
```

Figure 4-11 shows the resulting map.

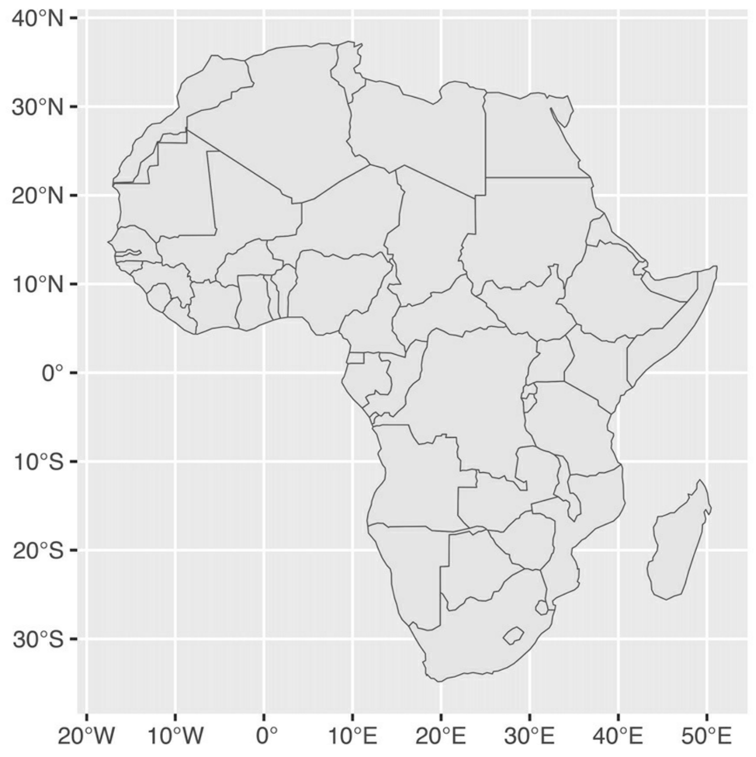


Figure 4-11: A map of Africa made with data from the rnaturalearth package

If you can’t find an appropriate package, you can always fall back on using `read_sf()` from the `sf` package.

Using Appropriate Projections

Once you have access to geospatial data, you need to decide which projection to use. If you’re looking for a simple answer to this question, you’ll be disappointed. As *Geocomputation with R* puts it, “The question of *which* CRS [to use] is tricky, and there is rarely a ‘right’ answer.”

If you’re overwhelmed by the task of choosing a projection, the `crsuggest` package from Kyle Walker can give you ideas. Its `suggest_top_crs()` function returns a coordinate reference system that is well suited for your data. Load `crsuggest` and try it out on your `africa_countries` data:

```
library(crsuggest)

africa_countries %>%
  suggest_top_crs()
```

The `suggest_top_crs()` function should return projection number `28232`. Pass this value to the `st_transform()` function to change the projection before you plot:

```
africa_countries %>%
  st_transform(28232) %>%
  ggplot() +
  geom_sf()
```

When run, this code generates the map in Figure 4-12.

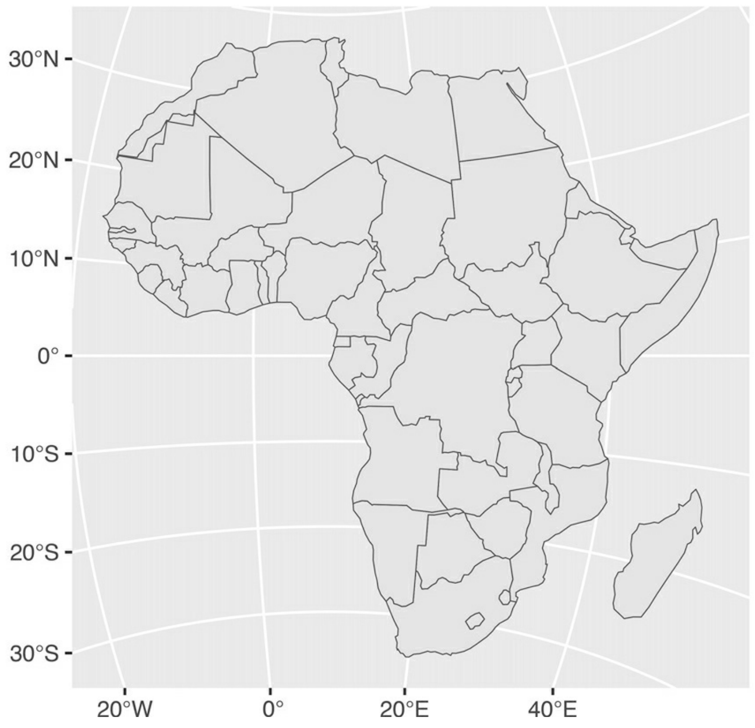


Figure 4-12: A map of Africa made with projection number 28232

As you can see, you've successfully mapped Africa with a different projection.

Wrangling Geospatial Data

The ability to merge traditional data frames with geospatial data is a huge benefit of working with simple features data. Remember that for his COVID-19 map, Madjid analyzed traditional data frames before merging them with geospatial data. But because simple features data acts just like traditional data frames, you can just as easily apply the data-wrangling and analysis functions from the `tidyverse` directly to a simple features object. To see how this works, revisit the `africa_countries` simple features data and select two variables (`name` and `pop_est`) to see the name and population of the countries:

```
africa_countries %>%
  select(name, pop_est)
```

The output looks like the following:

```
#> Simple feature collection with 51 features and 2 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -17.62504 ymin: -34.81917 xmax: 51.13
387 ymax: 37.34999
#> CRS:           +proj=longlat +datum=WGS84 +no_defs +ellps=
WGS84 +towgs84=0,0,0
#> First 10 features:
#>    name                     pop_est
#>  1 Angola                   12799293
#> 11 Burundi                  8988091
#> 13 Benin                    8791832
#> 14 Burkina Faso             15746232
#> 25 Botswana                 1990876
#> 26 Central African Rep.     4511488
#> 31 Côte d'Ivoire            20617068
#> 32 Cameroon                 18879301
#> 33 Dem. Rep. Congo          68692542
#> 34 Congo                    4012809
#>    geometry
#>  1 MULTIPOLYGON (((16.32653 -5...
#> 11 MULTIPOLYGON (((29.34 -4.49...
#> 13 MULTIPOLYGON (((2.691702 6....
#> 14 MULTIPOLYGON (((-2.827496 9...
#> 25 MULTIPOLYGON (((25.64916 -1...
#> 26 MULTIPOLYGON (((15.27946 7....
#> 31 MULTIPOLYGON (((-2.856125 4...
#> 32 MULTIPOLYGON (((13.07582 2....
#> 33 MULTIPOLYGON (((30.83386 3....
#> 34 MULTIPOLYGON (((12.99552 -4...
```

Say you want to make a map showing which African countries have populations larger than 20 million. First, you'll need to calculate this value for each country. To do so, use the `mutate()` and `if_else()` functions, which will return `TRUE` if a country's population is over 20 million and `FALSE` otherwise, and then store the result in a variable called `population_above_20_million`:

```
africa_countries %>%
  select(name, pop_est) %>%
  mutate(population_above_20_million = if_else(pop_est > 2000
0000, TRUE, FALSE))
```

You can then take this code and pipe it into ggplot, setting the `fill` aesthetic property to be equal to `population_above_20_million`:

```
africa_countries %>%
  select(name, pop_est) %>%
  mutate(population_above_20_million = if_else(pop_est > 2000
0000, TRUE, FALSE)) %>%
  ggplot(aes(fill = population_above_20_million)) +
  geom_sf()
```

This code generates the map shown in Figure 4-13.

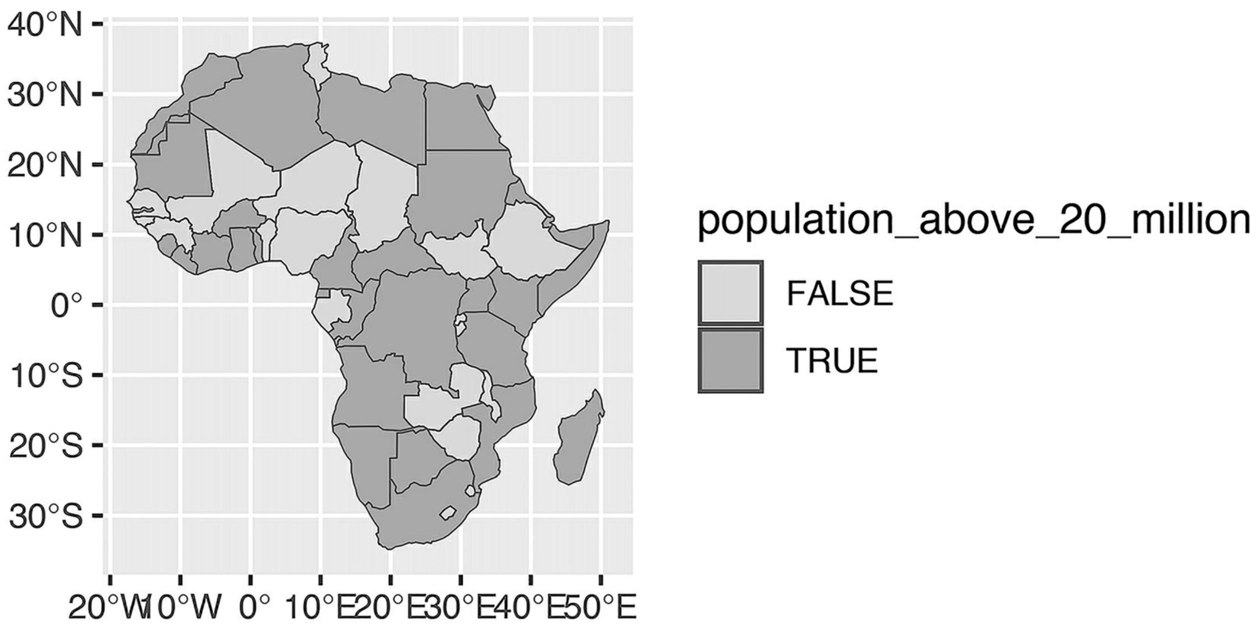


Figure 4-13: A map of Africa highlighting countries with populations above 20 million people

This is a basic example of the data wrangling and analysis you can perform on simple features data. The larger lesson is this: any skill you've developed for working with data in R will serve you well when working with geospatial data.

Summary

In this short romp through the world of mapmaking in R, you learned the basics of simple features geospatial data, reviewed how Abdoul Madjid applied this knowledge to make his map, explored how to get your own geospatial data, and saw how to project it appropriately to make your own maps.

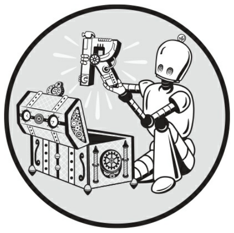
R may very well be the best tool for making maps. It also lets you use the skills you've developed for working with traditional data frames and the ggplot code to make your visualizations look great. After all, Madjid isn't a GIS expert, but he combined a basic understanding of geospatial data, fundamental R skills, and knowledge of data visualization principles to make a beautiful map. Now it's your turn to do the same.

Additional Resources

- Kieran Healy, "Draw Maps," in *Data Visualization: A Practical Introduction* (Princeton, NJ: Princeton University Press, 2018), *https://socviz.co*.
- Andrew Heiss, "Lessons on Space from Data Visualization: Use R, ggplot2, and the Principles of Graphic Design to Create Beautiful and Truthful Visualizations of Data," online course, last updated July 11, 2022, *https://datavizs22.classes.andrewheiss.com/content/12-content/*.
- Robin Lovelace, Jakub Nowosad, and Jannes Muenchow, *Geocomputation with R* (Boca Raton, FL: CRC Press, 2019), *https://r.geocompx.org*.
- Kyle Walker, *Analyzing US Census Data: Methods, Maps, and Models in R* (Boca Raton, FL: CRC Press, 2013).

5

DESIGNING EFFECTIVE TABLES



In his book *Fundamentals of Data Visualization*, Claus Wilke writes that tables are “an important tool for visualizing data.” This statement might seem odd. Tables are often considered the opposite of data visualizations such as plots: a place to dump numbers for the few nerds who care to read them. But Wilke sees things differently.

Tables need not—and should not—be data dumps devoid of design. While bars, lines, and points in graphs are visualizations, so are numbers in a table, and we should care about their appearance. As an example, take a look at the tables made by reputable news sources; data dumps these are not. Media organizations, whose job it is to communicate effectively, pay a lot of attention to table design. But elsewhere, because of their apparent simplicity, Wilke writes, “[tables] may not always receive the attention they need.”

Many people use Microsoft Word to make tables, a strategy that has potential pitfalls. Wilke found that his version of Word included 105 built-in table styles. Of those, around 80 percent, including the default style, violated some key principle of table design. The good news is that R is a great tool for making high-quality tables. It has a number of packages for this purpose and, within these packages, several functions designed to make sure your tables follow important design principles.

Moreover, if you’re writing reports in R Markdown (which you’ll learn about in Chapter 6), you can include code that will generate a table when you export your document. By working with a single tool to create tables, text, and other visualizations, you won’t have to copy and paste your data, lowering the risk of human error.

This chapter examines table design principles and shows you how to apply them to your tables using R’s `gt` package, one of the most popular table-making packages (and, as you’ll soon see, one that uses good design principles by default). These principles, and the code in this chapter, are adapted from Tom Mock’s blog post “10+ Guidelines for Better Tables in R.” Mock works at Posit, the company that makes RStudio, and has become something of an R table connoisseur. This chapter walks you through examples of Mock’s code to show you how small tweaks can make a big difference.

Creating a Data Frame

You will begin by creating a data frame that you can use to make tables throughout this chapter. First, load the packages you need (the `tidyverse` for general data manipulation functions, `gapminder` for the data you’ll use, `gt` to make the tables, and `gtExtras` to do some table formatting):

```
library(tidyverse)
library(gapminder)
library(gt)
library(gtExtras)
```

As you saw in Chapter 2, the `gapminder` package provides country-level demographic statistics. To make a data frame for your table, you’ll use just a few countries (the first four, in alphabetical order: Afghanistan, Albania, Algeria, and Angola) and three years (1952, 1972, and 1992). The `gapminder` data has many years, but these will suffice to demonstrate table-making principles. The following code creates a data frame called `gdp`:

```
gdp <- gapminder %>%
  filter(country %in% c("Afghanistan", "Albania", "Algeria",
"Angola")) %>%
  select(country, year, gdpPercap) %>%
  mutate(country = as.character(country)) %>%
  pivot_wider(
    id_cols = country,
    names_from = year,
    values_from = gdpPercap
  ) %>%
  select(country, `1952`, `1972`, `1992`) %>%
  rename(Country = country)
```

Here's what `gdp` looks like:

```
#> # A tibble: 4 × 4
#>   Country     `1952` `1972` `1992`
#>   <chr>        <dbl>  <dbl>  <dbl>
#> 1 Afghanistan   779.   740.   649.
#> 2 Albania      1601.  3313.  2497.
#> 3 Algeria      2449.  4183.  5023.
#> 4 Angola       3521.  5473.  2628.
```

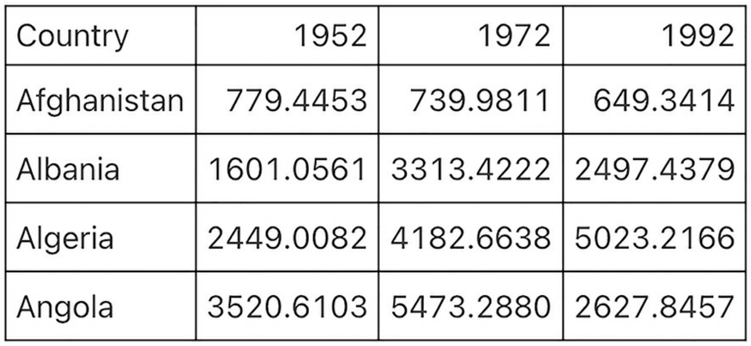
Now that you have some data, you'll use it to make a table.

Table Design Principles

Unsurprisingly, the principles of good table design are similar to those for data visualization more generally. This section covers six of the most important ones.

Minimize Clutter

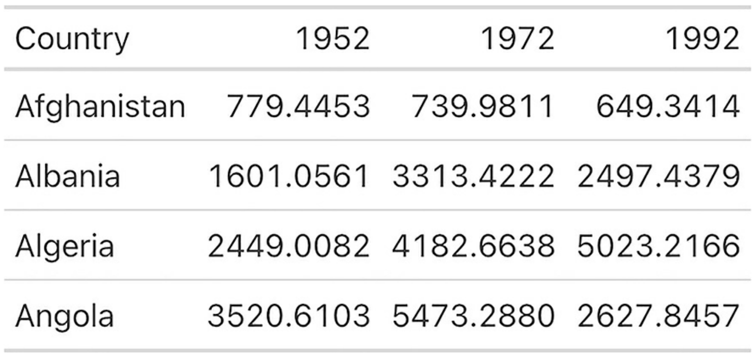
You can minimize clutter in your tables by removing unnecessary elements. For example, one common source of table clutter is grid lines, as shown in Figure 5-1.



Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

Figure 5-1: A table with grid lines everywhere can be distracting.

Having grid lines around every single cell in your table is unnecessary and distracts from the goal of communicating clearly. A table with minimal or even no grid lines (Figure 5-2) is a much more effective communication tool.



Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

Figure 5-2: A table with only horizontal grid lines is more effective.

I mentioned that `gt` uses good table design principles by default, and this is a great example. The second table, with minimal grid lines, requires just two lines of code—piping the `gdp` data into the `gt()` function, which creates a table:

```
gdp %>%
  gt()
```

To add grid lines to every part of the example, you'd have to add more code. Here, the code that follows the `gt()` function adds grid lines:

```
gdp %>%
  gt() %>%
  tab_style(
    style = cell_borders(
      side = "all",
      color = "black",
      weight = px(1),
      style = "solid"
    ),
    locations = list(
      cells_body(
        everything()
      ),
      cells_column_labels(
        everything()
      )
    )
  ) %>%
  opt_table_lines(extent = "none")
```

Since I don't recommend taking this approach, I won't walk you through this code. However, if you wanted to remove additional grid lines, you could do so like this:

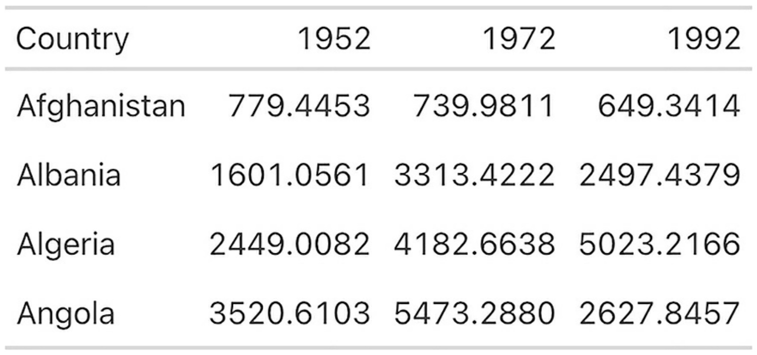
```
gdp %>%
  gt() %>%
  tab_style(
    style = cell_borders(color = "transparent"),
    locations = cells_body()
  )
```

The `tab_style()` function uses a two-step approach. First, it identifies the style to modify (in this case, the borders), then it specifies where to apply these modifications. Here, `tab_style()` tells R to modify the borders using the `cell_borders()` function, making the borders transparent, and to apply this transformation to the `cells_body()` location (versus, say, the `cells_column_labels()` for only the first row).

NOTE

To see all options, check out the list of so-called helper functions on the gt *package documentation website at* https://gt.rstudio.com/reference/index.xhtml#helper-functions.

Running this code outputs a table with no grid lines at all in the body (Figure 5-3).



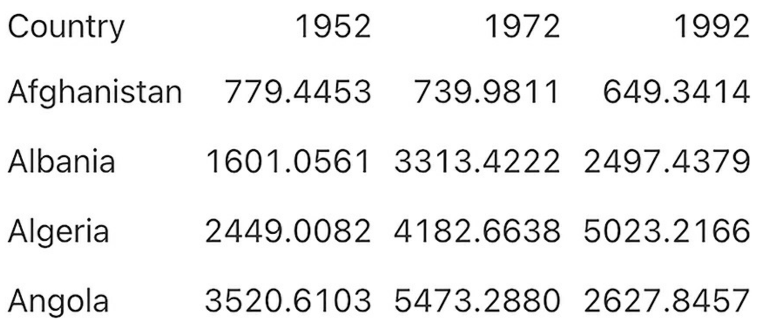
Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

Figure 5-3: A clean-looking table with grid lines only on the header row and the bottom

Save this table as an object called `table_no_gridlines` so that you can add to it later.

Differentiate the Header from the Body

While reducing clutter is an important goal, going too far can have negative consequences. A table with no grid lines at all can make it hard to differentiate between the header row and the table body. Consider Figure 5-4, for example.



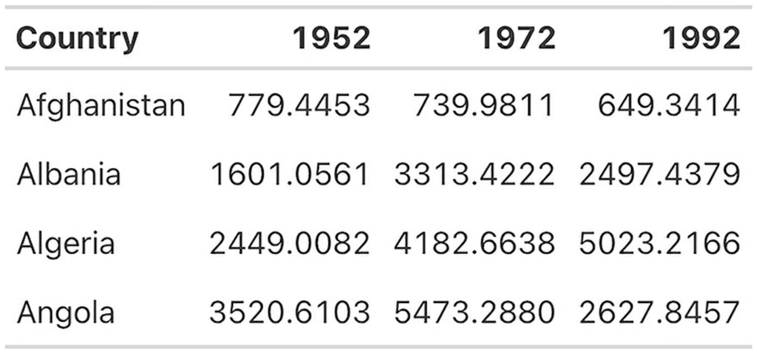
Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

Figure 5-4: An unclear table with all grid lines removed

By making the header row bold, you can make it stand out better:

```
table_no_gridlines %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_labels()
  )
```

Starting with the `table_no_gridlines` object, this code applies formatting with the `tab_style()` function in two steps. First, it specifies that it wants to alter the text style by using the `cell_text()` function to set the weight to bold. Second, it sets the location for this transformation to the header row using the `cells_column_labels()` function. Figure 5-5 shows what the table looks like with its header row bolded.



Country	**1952**	**1972**	**1992**
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

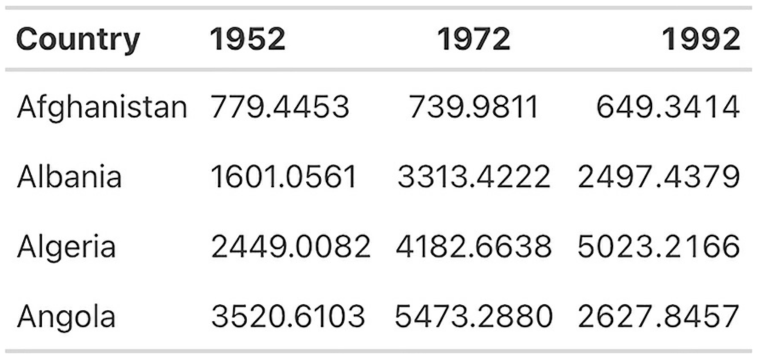
Figure 5-5: Making the header row more obvious using bold

Save this table as `table_bold_header` in order to add further formatting.

Align Appropriately

A third principle of high-quality table design is appropriate alignment. Specifically, numbers in tables should be right-aligned. Tom Mock explains that left-aligning or center-aligning numbers “impairs the ability to clearly compare numbers and decimal places. Right alignment lets you align decimal places and numbers for easy parsing.”

Let’s look at this principle in action. In Figure 5-6, the 1952 column is left-aligned, the 1972 column is center-aligned, and the 1992 column is right-aligned.



Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457

Figure 5-6: Comparing numerical data aligned to the left (1952), center (1972), and right (1992)

You can see how much easier it is to compare the values in the 1992 column than those in the other two columns. In both the 1952 and 1972 columns, it's challenging to compare the values because the numbers in the same position (the tens place, for example) aren't aligned vertically. In the 1992 column, however, the number in the tens place in Afghanistan (4) aligns with the number in the tens place in Albania (9) and all other countries, making it much easier to scan the table.

As with other tables, you actually have to override the defaults to get the `gt` package to misalign the columns, as demonstrated in the following code:

```
table_bold_header %>%
  cols_align(
    align = "left",
    columns = 2
  ) %>%
  cols_align(
    align = "center",
    columns = 3
  ) %>%
  cols_align(
    align = "right",
    columns = 4
  )
```

By default, `gt` will right-align numeric values. Don't change anything, and you'll be golden.

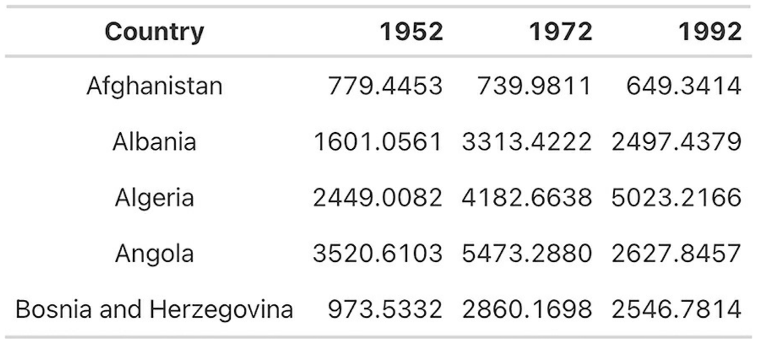
Right alignment is best practice for numeric columns, but for text columns, use left alignment. As Jon Schwabish points out in his article “Ten Guidelines for Better Tables” in the *Journal of Benefit-Cost Analysis*, it’s much easier to read longer text cells when they are left-aligned. To see the benefit of left-aligning text, add a country with a long name to your table. I’ve added Bosnia and Herzegovina and saved this as a data frame called `gdp_with_bosnia`. You’ll see that I’m using nearly the same code I used previously to create the `gdp` data frame:

```
gdp_with_bosnia <- gapminder %>%
  filter(country %in% c("Afghanistan", "Albania", "Algeria",
"Angola",
"Bosnia and Herzegovina")) %>%
  select(country, year, gdpPercap) %>%
  mutate(country = as.character(country)) %>%
  pivot_wider(
    id_cols = country,
    names_from = year,
    values_from = gdpPercap
  ) %>%
  select(country, `1952`, `1972`, `1992`) %>%
  rename(Country = country)
```

Here’s what the `gdp_with_bosnia` data frame looks like:

```
#> # A tibble: 5 × 4
#>   Country                `1952` `1972` `1992`
#>   <chr>                   <dbl>  <dbl>  <dbl>
#> 1 Afghanistan              779.   740.   649.
#> 2 Albania                 1601.  3313.  2497.
#> 3 Algeria                 2449.  4183.  5023.
#> 4 Angola                  3521.  5473.  2628.
#> 5 Bosnia and Herzegovina   974.  2860.  2547.
```

Now take the `gdp_with_bosnia` data frame and create a table with the Country column center-aligned. In the table in Figure 5-7, it’s hard to scan the country names, and that center-aligned column just looks a bit weird.



Country	1952	1972	1992
Afghanistan	779.4453	739.9811	649.3414
Albania	1601.0561	3313.4222	2497.4379
Algeria	2449.0082	4182.6638	5023.2166
Angola	3520.6103	5473.2880	2627.8457
Bosnia and Herzegovina	973.5332	2860.1698	2546.7814

Figure 5-7: Center-aligned text can be hard to read, especially when it includes longer values.

This is another example where you have to change the `gt` defaults to mess things up. In addition to right-aligning numeric columns by default, `gt` left-aligns character columns. As long as you don't touch anything, you'll get the alignment you're looking for.

If you ever do want to override the default alignments, you can use the `cols_align()` function. For example, here's how to make the table with center-aligned country names:

```
gdp_with_bosnia %>%
  gt() %>%
  tab_style(
    style = cell_borders(color = "transparent"),
    locations = cells_body()
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_labels()
  ) %>%
  cols_align(
    columns = "Country",
    align = "center"
  )
```

The `columns` argument tells `gt` which columns to align, and the `align` argument selects the alignment (`left`, `right`, or `center`).

Use the Correct Level of Precision

In all of the tables you've made so far, you've used the data exactly as it came to you. The data in the numeric columns, for example, extends to four decimal places—almost certainly too many. Having more decimal places makes a table harder to read, so you should always strike a balance between what Jon Schwabish describes as “necessary precision and a clean, spare table.”

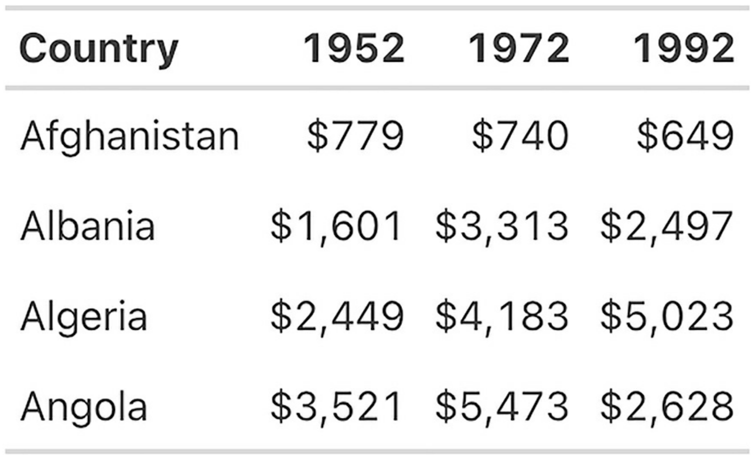
Here’s a good rule of thumb: if adding more decimal places would change some action, keep them; otherwise, take them out. In my experience, people tend to leave too many decimal places in, putting too much importance on a very high degree of accuracy (and, in the process, reducing the legibility of their tables).

In the GDP table, you can use the `fmt_currency()` function to format the numeric values:

```
table_bold_header %>%
  fmt_currency(
    columns = c(`1952`, `1972`, `1992`),
    decimals = 0
  )
```

The `gt` package has a whole series of functions for formatting values in tables, all of which start with `fmt_`. This code applies `fmt_currency()` to the 1952, 1972, and 1992 columns, then uses the `decimals` argument to tell `fmt_currency()` to format the values with zero decimal places. After all, the difference between a GDP of $779.4453 and $779 is unlikely to lead to different decisions.

This produces values formatted as dollars. The `fmt_currency()` function automatically adds a thousands-place comma to make the values even easier to read (Figure 5-8).



Country	1952	1972	1992
Afghanistan	$779	$740	$649
Albania	$1,601	$3,313	$2,497
Algeria	$2,449	$4,183	$5,023
Angola	$3,521	$5,473	$2,628

Figure 5-8: Rounding dollar amounts to whole numbers and adding dollar signs can simplify data.

Save your table for reuse as `table_whole_numbers`.

Use Color Intentionally

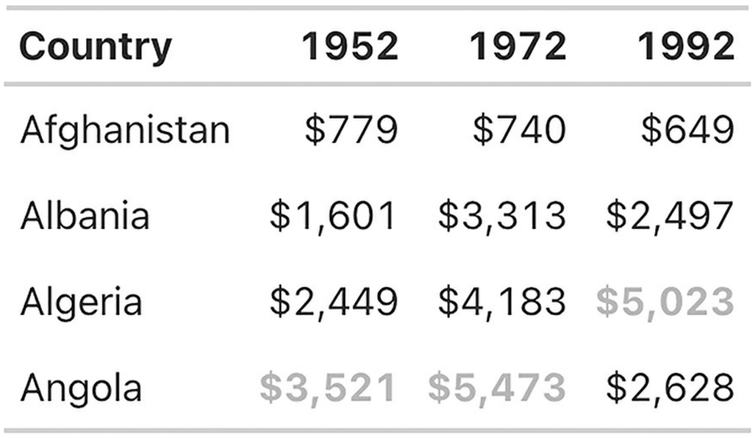
So far, your table hasn't used any color, so you'll add some now to highlight outlier values. Doing so can help your table communicate more effectively, especially for readers who want to scan it. To make the highest value in the year 1952 a different color, you again use the `tab_style()` function:

```
table_whole_numbers %>%
  tab_style(
    style = cell_text(
      color = "orange",
      weight = "bold"
    ),
    locations = cells_body(
      columns = `1952`,
      rows = `1952` == max(`1952`)
    )
  )
```

This function uses `cell_text()` to change the color of the text to orange and make it bold. Within the `cells_body()` function, the `locations()` function specifies the columns and rows to which the changes will apply. The `columns` argument is simply set to the year whose values are being changed, but setting the rows requires a more complicated formula. The code `rows =`

`` `1952` == max(`1952`) `` applies the text transformation to rows whose value is equal to the maximum value in that year.

Repeating this code for the 1972 and 1992 columns generates the result shown in Figure 5-9 (which represents the orange values in grayscale for print purposes).



Country	1952	1972	1992
Afghanistan	$779	$740	$649
Albania	$1,601	$3,313	$2,497
Algeria	$2,449	$4,183	$5,023
Angola	$3,521	$5,473	$2,628

Figure 5-9: Using color to highlight important values, such as the largest number in each year

The `gt` package makes it straightforward to add color to highlight outlier values.

Add a Data Visualization Where Appropriate

Adding color to highlight outliers is one way to help guide the reader's attention. Another way is to incorporate graphs into tables. Tom Mock developed an add-on package for `gt` called `gtExtras` that makes it possible to do just this. For example, say you want to show how the GDP of each country changes over time. To do that, you can add a new column that visualizes this trend using a *sparkline* (essentially, a simple line chart):

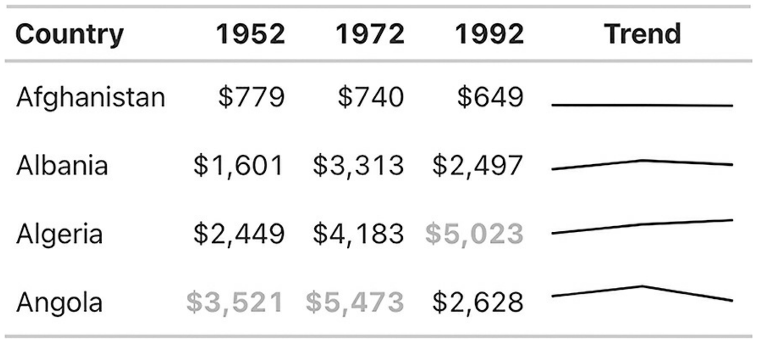
```
gdp_with_trend <- gdp %>%
  group_by(Country) %>%
  mutate(Trend = list(c(`1952`, `1972`, `1992`))) %>%
  ungroup()
```

The `gt_plt_sparkline()` function requires you to provide the values needed to make the sparkline in a single column. To accomplish this, the code creates a variable called `Trend`, using `group_by()` and `mutate()`, to hold a list of the values for each country. For Afghanistan, for example, `Trend` would contain 779.4453145, 739.9811058, and 649.3413952. Save this data as an object called `gdp_with_trend`.

Now you create your table as before but add the `gt_plt_sparkline()` function to the end of the code. Within this function, specify which column to use to create the sparkline (`Trend`) as follows:

```
gdp_with_trend %>%
  gt() %>%
  tab_style(
    style = cell_borders(color = "transparent"),
    locations = cells_body()
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_labels()
  ) %>%
  fmt_currency(
    columns = c(`1952`, `1972`, `1992`),
    decimals = 0
  ) %>%
  tab_style(
    style = cell_text(
      color = "orange",
      weight = "bold"
    ),
    locations = cells_body(
      columns = `1952`,
      rows = `1952` == max(`1952`)
    )
  ) %>%
  tab_style(
    style = cell_text(
      color = "orange",
      weight = "bold"
    ),
    locations = cells_body(
      columns = `1972`,
      rows = `1972` == max(`1972`)
    )
  ) %>%
  tab_style(
    style = cell_text(
      color = "orange",
      weight = "bold"
    ),
    locations = cells_body(
      columns = `1992`,
      rows = `1992` == max(`1992`)
    )
  ) %>%
  gt_plt_sparkline(
    column = Trend,
    label = FALSE,
    palette = c("black", "transparent", "transparent", "trans
parent", "transparent")
  )
```

Setting `label = FALSE` removes text labels that `gt_plt_sparkline()` adds by default, then adds a `palette` argument to make the sparkline black and all other elements of it transparent. (By default, the function will make different parts of the sparkline different colors.) The stripped-down sparkline in Figure 5-10 allows the reader to see the trend for each country at a glance.

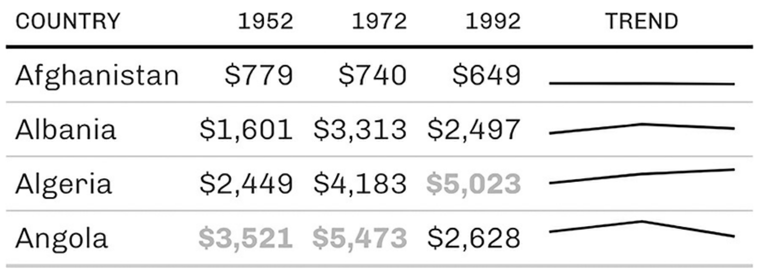


Country	1952	1972	1992	Trend
Afghanistan	$779	$740	$649	
Albania	$1,601	$3,313	$2,497	
Algeria	$2,449	$4,183	$5,023	
Angola	$3,521	$5,473	$2,628	

Figure 5-10: A table with sparklines can show changes in data over time.

The `gtExtras` package can do much more than merely create sparklines. Its set of theme functions allows you to make your tables look like those published by FiveThirtyEight, the *New York Times*, the *Guardian*, and other news outlets.

As an example, try removing the formatting you've applied so far and instead use the `gt_theme_538()` function to style the table. Then take a look at tables on the FiveThirtyEight website. You should see similarities to the one in Figure 5-11.



COUNTRY	1952	1972	1992	TREND
Afghanistan	$779	$740	$649	
Albania	$1,601	$3,313	$2,497	
Algeria	$2,449	$4,183	$5,023	
Angola	$3,521	$5,473	$2,628	

Figure 5-11: A table redone in the FiveThirtyEight style

Add-on packages like `gtExtras` are common in the table-making landscape. If you're working with the `reactable` package to make interactive tables, for example, you can also use the `reactablefmtr` to add interactive sparklines, themes, and more. You'll learn more about making interactive tables in Chapter 9.

Summary

Many of the tweaks you made to your table in this chapter are quite subtle. Changes like removing excess grid lines, bolding header text, right-aligning numeric values, and adjusting the level of precision can often go unnoticed, but if you skip them, your table will be far less effective. The final product isn't flashy, but it does communicate clearly.

You used the `gt` package to make your high-quality table, and as you've repeatedly seen, this package has good defaults built in. Often, you don't need to change much in your code to make effective tables. But no matter which package you use, it's essential to treat tables as worthy of just as much thought as other kinds of data visualization.

In Chapter 6, you'll learn how to create reports using R Markdown, which can integrate your tables directly into the final document. What's better than using just a few lines of code to make publication-ready tables?

Additional Resources

- Thomas Mock, "10+ Guidelines for Better Tables in R," *The MockUp*, September 4, 2020, *https://themockup.blog/posts/2020-09-04-10-table-rules-in-r/*.

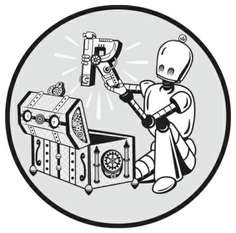
- Albert Rapp, “Creating Beautiful Tables in R with {gt},” November 27, 2022, *https://gt.albert-rapp.de*.
- Jon Schwabish, “Ten Guidelines for Better Tables,” *Journal of Benefit-Cost Analysis* 11, no. 2 (2020), *https://doi.org/10.1017/bca.2020.11*.

PART II

REPORTS, PRESENTATIONS, AND WEBSITES

6

R MARKDOWN REPORTS



Imagine that you’ve collected surveys about customer satisfaction with your new product. Now you’re ready to analyze the data and write up your results. First, you download your data from Google Sheets and import it into a statistical analysis tool like SPSS. Next, you use SPSS to clean and analyze your data, export summaries of your data as Excel spreadsheets, and then use Excel to make some charts. Finally, you write your report in Word, pasting in your charts from Excel along the way.

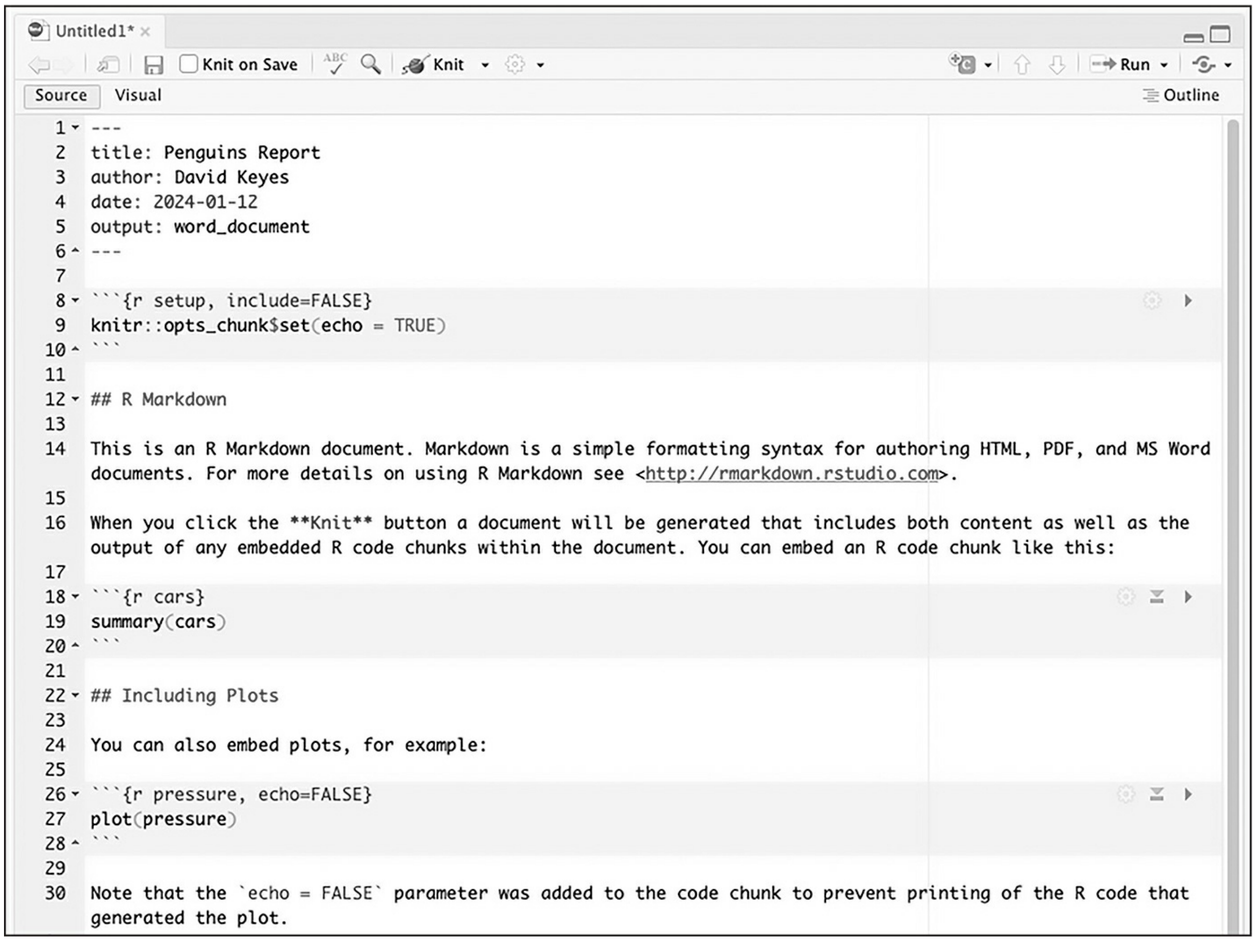
Sound familiar? If so, you’re not alone. Many people use this workflow for data analysis. But what happens when, the next month, new surveys roll in, and you have to redo your report? Yup, back through steps one through five. This multi-tool process might work for a one-time project, but let’s be honest: few projects are really one-time. For example, you might catch a mistake or realize you forgot to include a couple of surveys in your original analysis.

R Markdown combines data analysis, data visualization, and other R code with narrative text to create a document that can be exported to many formats, including Word, PDF, and HTML, to share with non-R users. When you use a single tool, your workflow becomes much more efficient. If you need to re-create that January customer satisfaction report in February, you can rerun your code to produce a new document with the newest data, and to fix an error in your analysis, you can simply adjust your code.

The ability to easily update reports at any time is known as *reproducibility*, and it's central to the value of R Markdown. This chapter breaks down the pieces of an R Markdown document, then describes some potential pitfalls and best practices. You'll learn how to work with YAML metadata, R code chunks, and Markdown-formatted text; create inline R code that can change the report's text dynamically; and run the document's code in various ways.

Creating an R Markdown Document

To create an R Markdown document in RStudio, go to **File ▸ New File ▸ R Markdown**. Choose a title, author, and date, as well as your default output format (HTML, PDF, or Word). These values can be changed later. Click **OK**, and RStudio will create an R Markdown document with some placeholder content, as shown in Figure 6-1.



~~~
---
title: Penguins Report
author: David Keyes
date: 2024-01-12
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word
documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as the
output of any embedded R code chunks within the document. You can embed an R code chunk like this:

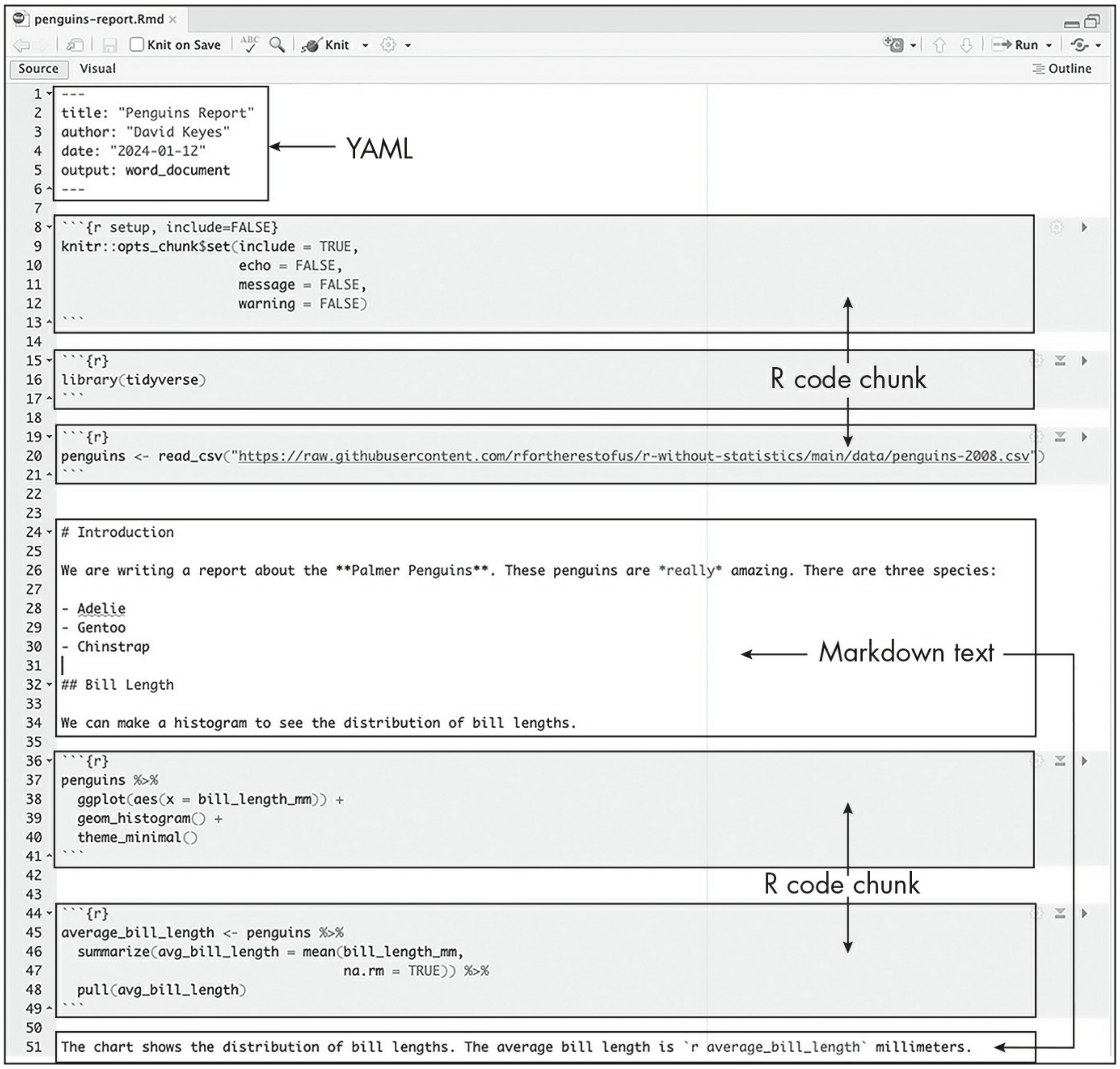
```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that
generated the plot.
```

Figure 6-1: The placeholder content in a new R Markdown document

The Knit menu at the top of RStudio converts an R Markdown document to the format you selected when creating it. In this example, the output format is set to be Word, so RStudio will create a Word document when you knit.

Delete the document’s placeholder content. In the next section, you’ll replace it with your own.

Document Structure

To explore the structure of an R Markdown document, you’ll create a report about penguins using data from the `palmerpenguins` package introduced in Chapter 3. I’ve separated the data by year, and you’ll use just the 2007 data. Figure 6-2 shows the complete R Markdown document, with boxes surrounding each section.
```



~~~
---
title: "Penguins Report"
author: "David Keyes"
date: "2024-01-12"
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(include = TRUE,
                      echo = FALSE,
                      message = FALSE,
                      warning = FALSE)
~~~

```{r}
library(tidyverse)
```

```{r}
penguins <- read_csv("https://raw.githubusercontent.com/rfortherestofus/r-without-statistics/main/data/penguins-2008.csv")
```

Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

Bill Length

We can make a histogram to see the distribution of bill lengths.

```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```

```{r}
average_bill_length <- penguins %>%
  summarize(avg_bill_length = mean(bill_length_mm,
                                   na.rm = TRUE)) %>%
  pull(avg_bill_length)
```

The chart shows the distribution of bill lengths. The average bill length is `r average_bill_length` millimeters.
```

*Figure 6-2: Components of an R Markdown document*

All R Markdown documents have three main parts: one YAML section, multiple R code chunks, and sections of Markdown text.

## *The YAML Metadata*

The YAML section is the very beginning of an R Markdown document. The name YAML comes from the recursive acronym *YAML ain't markup language*, whose meaning isn't important for our purposes. Three dashes indicate its beginning and end, and the text inside of it contains metadata
```

about the R Markdown document:

~~~~
---
title: Penguins Report
author: David Keyes
date: 2024-01-12
output: word_document
---
~~~~

As you can see, the YAML provides the title, author, date, and output format. All elements of the YAML are given in `key: value` syntax, where each key is a label for a piece of metadata (for example, the title) followed by the value.

The R Code Chunks

R Markdown documents have a different structure from the R script files you might be familiar with (those with the *.R* extension). R script files treat all content as code unless you comment out a line by putting a hash mark (`#`) in front of it. In the following listing, the first line is a comment, and the second line is code:

~~~~
# Import our data
data <- read_csv("data.csv")
~~~~

In R Markdown, the situation is reversed. Everything after the YAML is treated as text unless you specify otherwise by creating *code chunks*. These start with three backticks (```` ``` ````), followed by the lowercase letter `r` surrounded by curly brackets (`{}`). Another three backticks indicate the end of the code chunk:

~~~~
```{r}
library(tidyverse)
```
~~~~

If you're working in RStudio, code chunks should have a light gray background.

R Markdown treats anything in the code chunk as R code when you knit.

For example, this code chunk will produce a histogram in the final Word document:

~~~
```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

Figure 6-3 shows the resulting histogram.

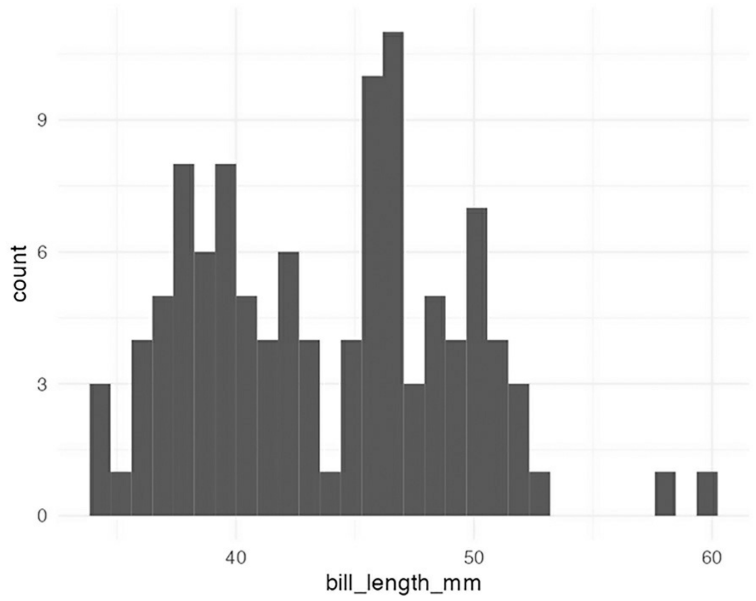


Figure 6-3: A simple histogram generated by an R Markdown code chunk

A code chunk at the top of each R Markdown document, known as the *setup code chunk*, gives instructions for what should happen when knitting a document. It contains the following options:

`echo` Do you want to show the code itself in the knitted document?

`include` Do you want to show the output of the code chunk?

`message` Do you want to include any messages that code might generate? For example, this message shows up when you run `library(tidyverse)`:

```
── Attaching core tidyverse packages ───── tidyverse 1.x.x──
✔ dplyr     1.x.x      ✔ readr    2.x.x
✔ forcats   0.x.x      ✔ stringr  1.x.x
✔ ggplot2   3.x.x      ✔ tibble   3.x.x
✔ lubridate 1.x.x      ✔ tidyr    1.x.x
✔ purrr     1.x.x
── Conflicts───── tidyverse_conflicts() ──
x dplyr::filter() masks stats::filter()
x dplyr::lag()    masks stats::lag()
```

`warning` Do you want to include any messages that the code might generate? For example, here's the message you get when creating a histogram using `geom_histogram()`:

```
`stat_bin()` using `bins = 30`. Pick better value with `binwi
dth`.
```

NOTE

To see the full list of code chunk options, visit https://yihui.org/knitr/options/.

In cases where you're using R Markdown to generate a report for a non-R user, you likely would want to hide the code, messages, and warnings but show the output (which would include any visualizations you generate). The following setup code chunk does this:

~~~
```{r setup, include = FALSE}
knitr::opts_chunk$set(include = TRUE,
                      echo = FALSE,
                      message = FALSE,
                      warning = FALSE)
```
~~~

The `include = FALSE` option on the first line applies to the setup code chunk itself. It tells R Markdown not to include the output of the setup code chunk when knitting. The options within `knitr::opts_chunk$set()` apply to all future code chunks. However, you can also override these global code chunk options on individual chunks. If you wanted your Word document to show both the plot itself and the code used to make it, for example, you could set `echo = TRUE` for that code chunk only:

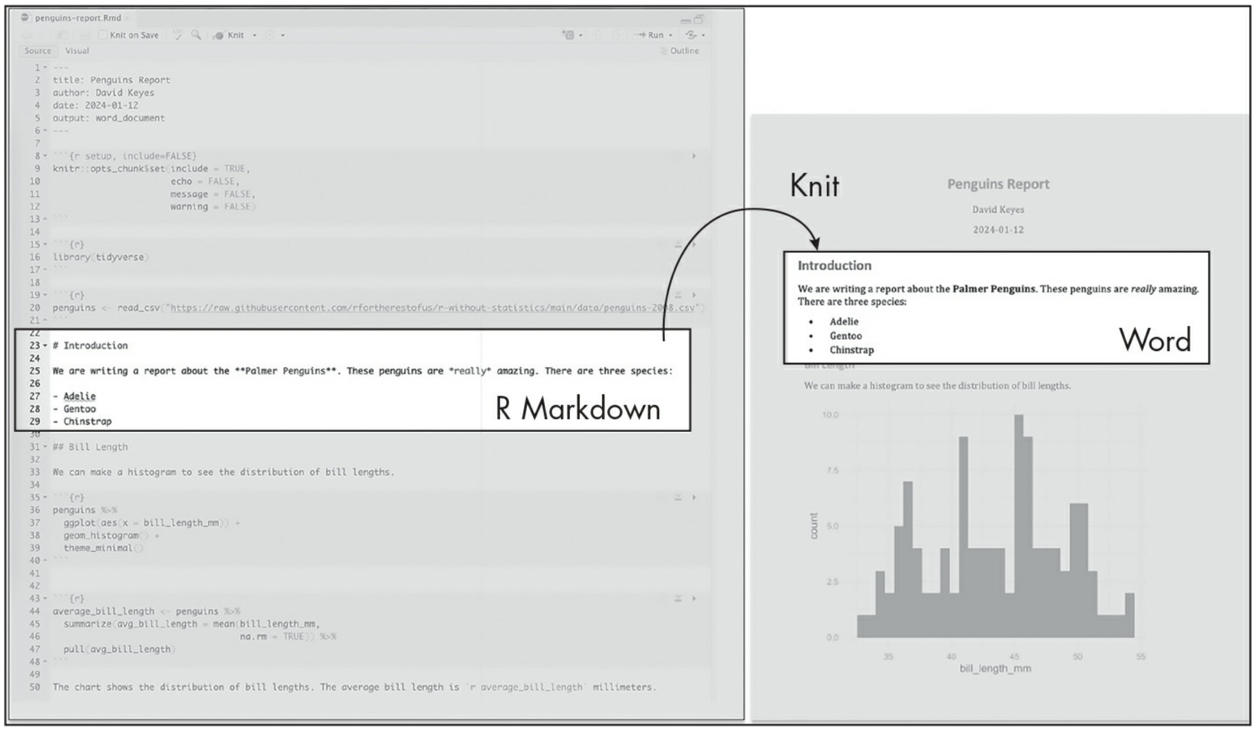
~~~
```{r echo = TRUE}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

Because `include` is already set to `TRUE` within `knitr::opts_chunk$set()` in the `setup` code chunk, you don't need to specify it again.

Markdown Text

Markdown is a way to style text. If you were writing directly in Word, you could just press the B button to make text bold, for example, but R doesn't have such a button. If you want your knitted Word document to include bold text, you need to use Markdown to indicate this style in the document.

Markdown text sections (which have a white background in RStudio) will be converted into formatted text in the Word document after knitting. Figure 6-4 highlights the equivalent sections in the R Markdown and Word documents.



~~~
---
title: Penguins Report
author: David Keyes
date: 2024-01-12
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(include = TRUE,
                      echo = FALSE,
                      message = FALSE,
                      warning = FALSE)
```

```{r}
library(tidyverse)
```

```{r}
penguins <- read_csv("https://raw.githubusercontent.com/rfortherestofus/r-without-statistics/main/data/penguins-2008.csv")
```

# Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

## Bill Length

We can make a histogram to see the distribution of bill lengths.

```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```


```{r}
average_bill_length <- penguins %>%
  summarize(avg_bill_length = mean(bill_length_mm,
                                   na.rm = TRUE)) %>%
  pull(avg_bill_length)
```

The chart shows the distribution of bill lengths. The average bill length is `r average_bill_length` millimeters.
~~~

Penguins Report

David Keyes

2024-01-12

Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

Bill Length

We can make a histogram to see the distribution of bill lengths.

Figure 6-4: Markdown text in R Markdown and its equivalent in a knitted Word document

The text `# Introduction` in R Markdown gets converted to a first-level heading, while `## Bill Length` becomes a second-level heading. By adding hashes, you can create up to six levels of headings. In RStudio, headings are easy to find because they show up in blue.

Text without anything before it becomes body text in Word. To create italic text, add single asterisks around it (`*like this*`). To make text bold, use double asterisks (`**as shown here**`).

You can make bulleted lists by placing a dash at the beginning of a line and adding your text after it:

```
- Adelie
- Gentoo
- Chinstrap
```

To make ordered lists, replace the dashes with numbers. You can either number each line consecutively or, as done below, repeat `1`. In the knitted document, the proper numbers will automatically generate.

```
1. Adelie
1. Gentoo
1. Chinstrap
```

Formatting text in Markdown might seem more complicated than doing so in Word. But if you want to switch from a multi-tool workflow to a reproducible R Markdown–based workflow, you need to remove all manual actions from the process so that you can easily repeat it in the future.

Inline R Code

R Markdown documents can also include little bits of code within Markdown text. To see how this inline code works, take a look at the following sentence in the R Markdown document:

```
The average bill length is `r average_bill_length` millimeter
s.
```

Inline R code begins with a backtick and the lowercase letter `r` and ends with another backtick. In this example, the code tells R to print the value of the variable `average_bill_length`, which is defined as follows in the code chunk before the inline code:

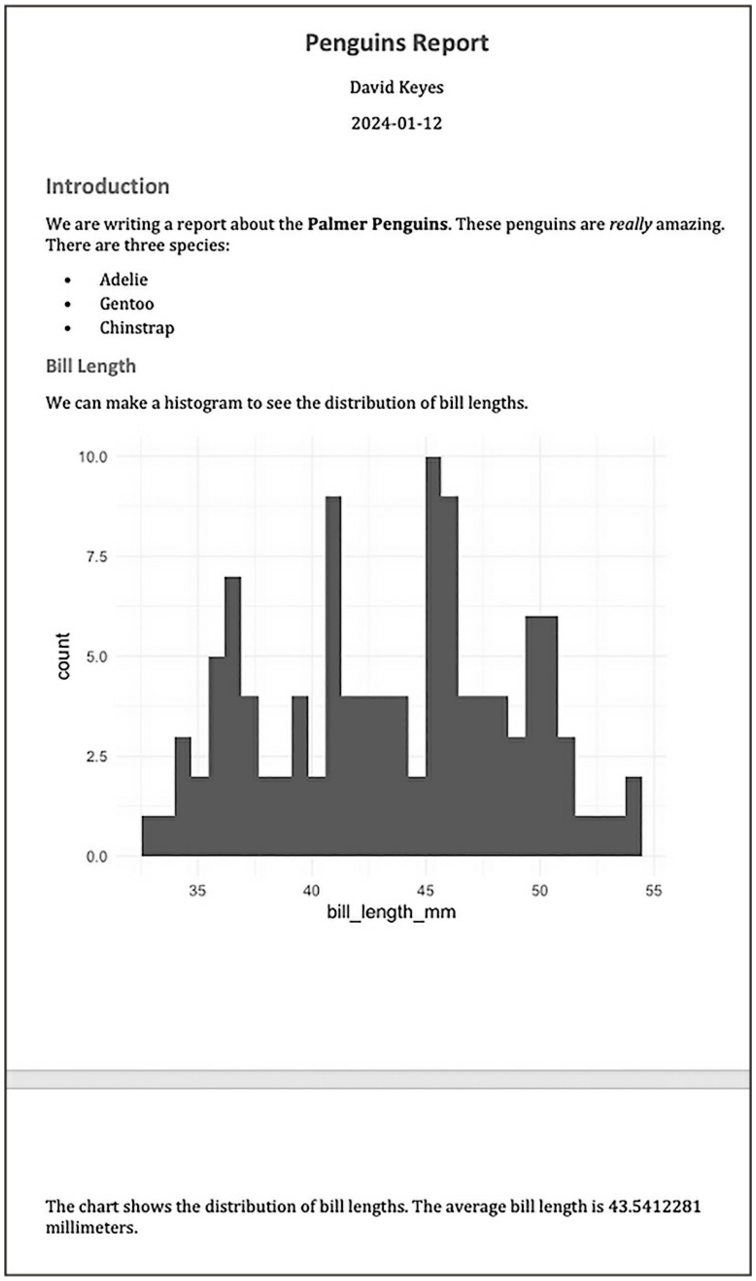
~~~
```{r}
average_bill_length <- penguins %>%
  summarize(avg_bill_length = mean(
    bill_length_mm,
    na.rm = TRUE
  )) %>%
  pull(avg_bill_length)
```
~~~

This code calculates the average bill length and saves it as `average_bill_length`. Having created this variable, you can now use it in the inline code. As a result, the Word document includes the sentence “The average bill length is 43.9219298.”

One benefit of using inline R code is that you avoid having to copy and paste values, which is error-prone. Inline R code also makes it possible to automatically calculate values on the fly whenever you reknit the R Markdown document with new data. To see how this works, you'll make a new report using data from 2008. To do this, you need to change only one line, the one that reads the data:

```
penguins <- read_csv("https://data.rfortherestofus.com/pengui
ns-2008.csv")
```

Now that you've switched *penguins-2007.csv* to *penguins-2008.csv*, you can reknit the report and produce a new Word document, complete with updated results. Figure 6-5 shows the new document.


Penguins Report

David Keyes

2024-01-12

Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

Bill Length

We can make a histogram to see the distribution of bill lengths.

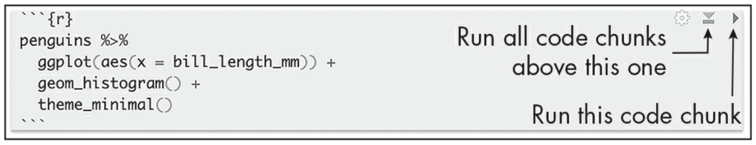
The chart shows the distribution of bill lengths. The average bill length is 43.5412281 millimeters.

Figure 6-5: The knitted Word document with 2008 data

The new histogram is based on the 2008 data, as is the average bill length of 43.5412281. These values update automatically because every time you press Knit, the code is rerun, regenerating plots and recalculating values. As long as the data you use has a consistent structure, updating a report requires just a click of the Knit button.

Running Code Chunks Interactively

You can run the code in an R Markdown document in two ways. The first is by knitting the entire document. The second is to run code chunks manually (also known as *interactively*) by pressing the green play button at the top right of a code chunk. The down arrow next to the green play button will run all code until that point. You can see these buttons in Figure 6-6.



~~~
```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

Figure 6-6: The buttons on code chunks in RStudio

You can also use COMMAND-ENTER on macOS or CTRL-ENTER on Windows to run sections of code, as in an R script file. Running code interactively is a good way to test that portions of it work before you knit the entire document.

The one downside to running code interactively is that you can sometimes make mistakes that cause your R Markdown document to fail to knit. That is because, in order to knit, an R Markdown document must contain all the code it uses. If you're working interactively and, say, load data from a separate file, you won't be able to knit your document. When working in R Markdown, always keep all your code within a single document.

The code must also appear in the right order. An R Markdown document that looks like this, for example, will give you an error if you try to knit it:

~~~
---
title: Penguins Report
author: David Keyes
date: 2024-01-12
output: word_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(
  include = TRUE,
  echo = FALSE,
  message = FALSE,
~~~

```
  warning = FALSE
)
```

```{r}
penguins <- read_csv("https://data.rfortherestofus.com/pengui
ns-2008.csv")
```

```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```

```{r}
library(tidyverse)
```
````

This error happens because you are attempting to use `tidyverse` functions like `read_csv()`, as well as various ggplot functions, before you load the `tidyverse` package.

Alison Hill, a research scientist and one of the most prolific R Markdown educators, tells her students to knit early and often. This practice makes it easier to isolate issues that make knitting fail. Hill describes her typical R Markdown workflow as spending 75 percent of her time working on a new document and 25 percent of her time knitting to check that the R Markdown document works.

Quarto

In 2022, Posit released a publishing tool similar to R Markdown. Known as Quarto, this tool takes what R Markdown has done for R and extends it to other languages, including Python, Julia, and Observable JS. As I write this book, Quarto is gaining traction. Luckily, the concepts you've learned in this chapter apply to Quarto as well. Quarto documents have a YAML section, code chunks, and Markdown text. You can export Quarto documents to HTML, PDF, and Word. However, R Markdown and Quarto documents have some syntactic differences, which are explored further in Chapter 10.
````

## Summary

You started this chapter by considering the scenario of a report that needs to be regenerated monthly. You learned how you can use R Markdown to reproduce this report every month without changing your code. Even if you lost the final Word document, you could quickly re-create it.

Best of all, working with R Markdown makes it possible to do in seconds what would have previously taken hours. When making a single report requires three tools and five steps, you may not want to work on it. But, as Alison Hill has pointed out, with R Markdown you can even work on reports before you receive all of the data. You could simply write code that works with partial data and rerun it with the final data at any time.

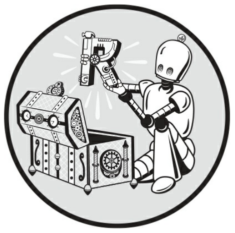
This chapter has just scratched the surface of what R Markdown can do. The next chapter will show you how to use it to instantly generate hundreds of reports. Magic indeed!

## Additional Resources

- Yihui Xie, J. J. Allaire, and Garrett Grolemund, *R Markdown: The Definitive Guide* (Boca Raton, FL: CRC Press, 2019), *https://bookdown.org/yihui/rmarkdown/*.
- Yihui Xie, Christophe Dervieux, and Emily Riederer, *R Markdown Cookbook* (Boca Raton, FL: CRC Press, 2021), *https://bookdown.org/yihui/rmarkdown-cookbook/*.

# 7

# PARAMETERIZED REPORTING



*Parameterized reporting* is a technique that allows you to generate multiple reports simultaneously. By using parameterized reporting, you can follow the same process to make 3,000 reports as you would to make one report. The technique also makes your work more accurate, as it avoids copy-and-paste errors.

Staff at the Urban Institute, a think tank based in Washington, DC, used parameterized reporting to develop fiscal briefs for all US states, as well as the District of Columbia. Each report required extensive text and multiple charts, so creating them by hand wasn't feasible. Instead, employees Safia Sayed, Livia Mucciolo, and Aaron Williams automated the process. This chapter explains how parameterized reporting works and walks you through a simplified version of the code that the Urban Institute used.

## Report Templates in R Markdown

If you've ever had to create multiple reports at the same time, you know how frustrating it can be, especially if you're using the multi-tool workflow described in Chapter 6. Making just one report can take a long time. Multiply that work by 10, 20, or, in the case of the Urban Institute team, 51, and it can quickly feel overwhelming. Fortunately, with parameterized reporting, you can generate thousands of reports at once using the following workflow:

1. Make a report template in R Markdown.
2. Add a parameter (for example, one representing US states) in the YAML of your R Markdown document to represent the values that will change between reports.
3. Use that parameter to generate a report for one state, to make sure you can knit your document.
4. Create a separate R script file that sets the value of the parameter and then knits a report.
5. Run this script for all states.

You'll begin by creating a report template for one state. I've taken the code that the Urban Institute staff used to make their state fiscal briefs and simplified it significantly. All of the packages used are ones you've seen in previous chapters, with the exception of the `urbnthemes` package. This package contains a custom ggplot theme. It can be installed by running `remotes::install_github("UrbanInstitute/urbnthemes")` in the console. Instead of focusing on fiscal data, I've used data you may be more familiar with: COVID-19 rates from mid-2022. Here's the R Markdown document:

~~~
---
title: "Urban Institute COVID Report"
output: html_document
params:
  state: "Alabama"
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(
  echo = FALSE,
  warning = FALSE,
  message = FALSE
)
```

```{r}
library(tidyverse)
library(urbnthemes)
~~~

```
library(scales)
```
```

```
# `r params$state`

```{r}
cases <- tibble(state.name) %>%
  rbind(state.name = "District of Columbia") %>%
  left_join(
    read_csv(
      "https://data.rfortherestofus.com/united_states_covid19
_cases_deaths_and_testing_by_state.csv",
      skip = 2
    ),
    by = c("state.name" = "State/Territory")
  ) %>%
  select(
    total_cases = `Total Cases`,
    state.name,
    cases_per_100000 = `Case Rate per 100000`
  ) %>%
  mutate(cases_per_100000 = parse_number(cases_per_100000)) %
>%
  mutate(case_rank = rank(-cases_per_100000, ties.method = "m
in"))
```
```

````
```{r}
state_text <- if_else(params$state == "District of Columbia",
 str_glue(
"the District of Columbia"), str_glue("state of {params$state
}"))

state_cases_per_100000 <- cases %>%
  filter(state.name == params$state) %>%
  pull(cases_per_100000) %>%
  comma()

state_cases_rank <- cases %>%
  filter(state.name == params$state) %>%
  pull(case_rank)
```

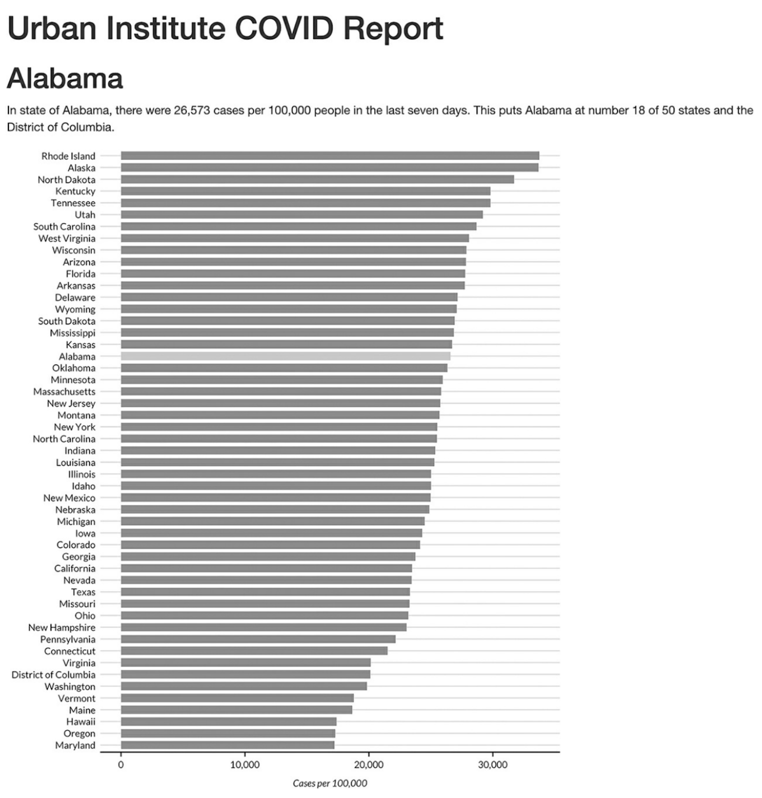
In `r state_text`, there were `r state_cases_per_100000` case
s per 100,000
````
````

~~~
people in the last seven days. This puts `r params$state` at
number
`r state_cases_rank` of 50 states and the District of Columbi
a.

```{r fig.height = 8}
set_urbn_defaults(style = "print")

cases %>%
  mutate(highlight_state = if_else(state.name == params$state
, "Y", "N")) %>%
  mutate(state.name = fct_reorder(state.name, cases_per_10000
0)) %>%
  ggplot(aes(
    x = cases_per_100000,
    y = state.name,
    fill = highlight_state
  )) +
  geom_col() +
  scale_x_continuous(labels = comma_format()) +
  theme(legend.position = "none") +
  labs(
    y = NULL,
    x = "Cases per 100,000"
  )
```
~~~

The text and charts in the report come from the `cases` data frame, shown here:

```
#> # A tibble: 51 × 4
#>    total_cases state.name  cases_per_100000 case_rank
#>    <chr>       <chr>                  <dbl>     <int>
#>  1 1302945     Alabama                26573        18
#>  2 246345      Alaska                 33675         2
#>  3 2025435     Arizona                27827        10
#>  4 837154      Arkansas               27740        12
#>  5 9274208     California             23472        35
#>  6 1388702     Colorado               24115        33
#>  7 766172      Connecticut            21490        42
#>  8 264376      Delaware               27150        13
#>  9 5965411     Florida                27775        11
#> 10 2521664     Georgia                23750        34
#> # ... with 41 more rows
```

When you knit the document, you end up with the simple HTML file shown in Figure 7-1.


Urban Institute COVID Report

Alabama

In state of Alabama, there were 26,573 cases per 100,000 people in the last seven days. This puts Alabama at number 18 of 50 states and the District of Columbia.

Figure 7-1: The Alabama report generated via R Markdown

You should recognize the R Markdown document’s YAML, R code chunks, inline code, and Markdown text from Chapter 6.

Defining Parameters

In R Markdown, *parameters* are variables that you set in the YAML to allow you to create multiple reports. Take a look at these two lines in the YAML:

```
params:
  state: "Alabama"
```

This code defines a variable called `state`. You can use the `state` variable throughout the rest of the R Markdown document with the `params$`*`variable_name`* syntax, replacing *`variable_name`* with `state` or any other name you set in the YAML. For example, consider this inline R code:

```
# `r params$state`
```

Any instance of the `params$state` parameter will be converted to `"Alabama"` when you knit it. This parameter and several others appear in the following code, which sets the first-level heading visible in Figure 7-1:

```
In `r state_text`, there were `r state_cases_per_100000` case
s per 100,000
people in the last seven days. This puts `r params$state` at
number
`r state_cases_rank` of 50 states and the District of Columbi
a.
```

After knitting the document, you should see the following text:

In the state of Alabama, there were 26,573 cases per 100,000 people in the last seven days. This puts Alabama at number 18 of 50 states and the District of Columbia.

This text is automatically generated. The inline R code `` `r state_text` `` prints the value of the variable `state_text`, which is determined by a previous call to `if_else()`, shown in this code chunk:

```
state_text <- if_else(params$state == "District of Columbia",
str_glue("the District of Columbia"), str_glue("state of {par
ams$state}"))
```

If the value of `params$states` is `"District of Columbia"`, this code sets `state_text` equal to `"the District of Columbia"`. If `params$state` isn't `"District of Columbia"`, then `state_text` gets the value `"state of"`, followed by the state name. This allows you to put `state_text` in a sentence and have it work no matter whether the state parameter is a state or the District of Columbia.

Generating Numbers with Parameters

You can also use parameters to generate numeric values to include in the text. For example, to calculate the values of the `state_cases_per_100000` and `state_cases_rank` variables dynamically, use the state parameter, as shown here:

```
state_cases_per_100000 <- cases %>%
  filter(state.name == params$state) %>%
  pull(cases_per_100000) %>%
  comma()

state_cases_rank <- cases %>%
  filter(state.name == params$state) %>%
  pull(case_rank)
```

First, this code filters the `cases` data frame (which contains data for all states) to keep only the data for the state in `params$state`. Then, the `pull()` function gets a single value from that data, and the `comma()` function from the `scales` package applies formatting to make `state_cases_per_100000` display as 26,573 (rather than 26573). Finally, the `state_cases_per_100000` and `state_case_rank` variables are integrated into the inline R code.

Including Parameters in Visualization Code

The `params$state` parameter is used in other places as well, such as to highlight a state in the report's bar chart. To see how to accomplish this, look at the following section from the last code chunk:

```
cases %>%
  mutate(highlight_state = if_else(state.name == params$state
, "Y", "N"))
```

This code creates a variable called `highlight_state`. Within the `cases` data frame, the code checks whether `state.name` is equal to `params$state`. If it is, `highlight_state` gets the value `Y`. If not, it gets `N`. Here's what the relevant columns look like after you run these two lines:

```
#> # A tibble: 51 × 2
#>    state.name  highlight_state
#>    <chr>       <chr>
#>  1 Alabama     Y
#>  2 Alaska      N
#>  3 Arizona     N
#>  4 Arkansas    N
#>  5 California  N
#>  6 Colorado    N
#>  7 Connecticut N
#>  8 Delaware    N
#>  9 Florida     N
#> 10 Georgia     N
#> #  ... with 41 more rows
```

Later, the ggplot code uses the `highlight_state` variable for the bar chart's `fill` aesthetic property, highlighting the state in `params$state` in yellow and coloring the other states blue. Figure 7-2 shows the chart with Alabama highlighted.

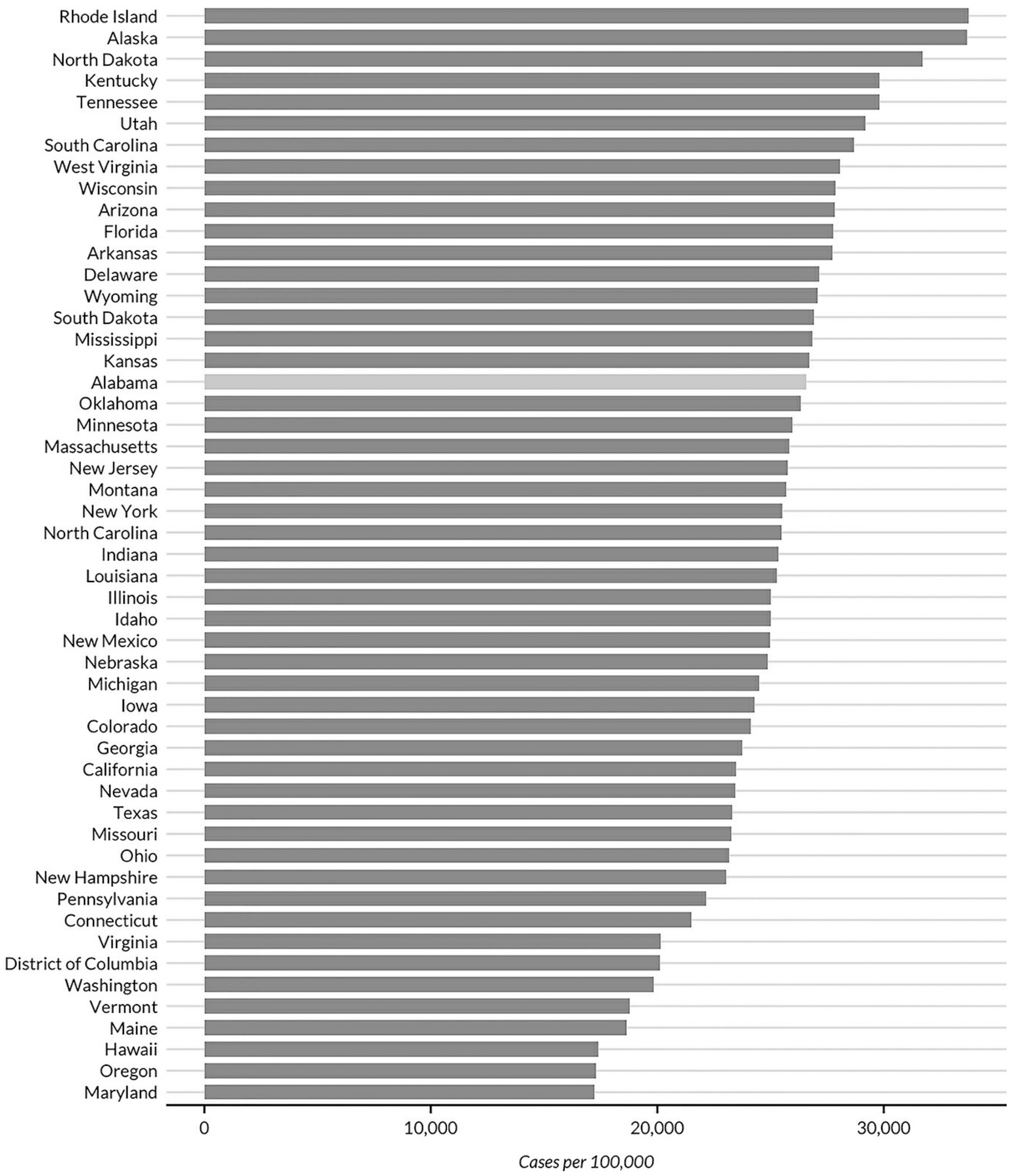


Figure 7-2: Highlighting data in a bar chart using parameters

As you've seen, setting a parameter in the YAML allows you to dynamically generate text and charts in the knitted report. But you've generated only one report so far. How can you create all 51 reports? Your first thought might be to manually update the YAML by changing the parameter's value from `"Alabama"` to, say, `"Alaska"` and then knitting the document again. While you *could* follow this process for all states, it would be tedious, which is what you're trying to avoid. Instead, you can automate the report generation.

Creating an R Script

To automatically generate multiple reports based on the template you've created, you'll use an R script that changes the value of the parameters in the R Markdown document and then knits it. You'll begin by creating an R script file named *render.R*.

Knitting the Document with Code

Your script needs to be able to knit an R Markdown document. While you've seen how to do this using the Knit button, you can do the same thing with code. Load the `rmarkdown` package and then use its `render()` function as shown here:

```
library(rmarkdown)

render(
  input = "urban-covid-budget-report.Rmd",
  output_file = "Alaska.xhtml",
  params = list(state = "Alaska")
)
```

This function generates an HTML document called *urban-covid-budget-report.xhtml*. By default, the generated file has the same name as the R Markdown (*.Rmd*) document, with a different extension. The `output_file` argument assigns the file a new name, and the `params` argument specifies parameters that will override those in the R Markdown document itself. For example, this code tells R to use Alaska for the `state` parameter and save the resulting HTML file as *Alaska.xhtml*.

This approach to generating reports works, but to create all 51 reports, you'd have to manually change the state name in the YAML and update the `render()` function before running it for each report. In the next section, you'll update your code to make it more efficient.

Creating a Tibble with Parameter Data

To write code that generates all your reports automatically, first you must create a *vector* (in colloquial terms, a list of items) of all the state names and the District of Columbia. To do this, you'll use the built-in dataset `state.name`, which has all 50 state names in a vector:

```
state <- tibble(state.name) %>%
  rbind("District of Columbia") %>%
  pull(state.name)
```

This code turns `state.name` into a tibble and then uses the `rbind()` function to add the District of Columbia to the list. The `pull()` function gets one single column and saves it as `state`. Here's what the `state` vector looks like:

```
#>  [1] "Alabama"              "Alaska"
#>  [3] "Arizona"              "Arkansas"
#>  [5] "California"           "Colorado"
#>  [7] "Connecticut"          "Delaware"
#>  [9] "Florida"              "Georgia"
#> [11] "Hawaii"               "Idaho"
#> [13] "Illinois"             "Indiana"
#> [15] "Iowa"                 "Kansas"
#> [17] "Kentucky"             "Louisiana"
#> [19] "Maine"                "Maryland"
#> [21] "Massachusetts"        "Michigan"
#> [23] "Minnesota"            "Mississippi"
#> [25] "Missouri"             "Montana"
#> [27] "Nebraska"             "Nevada"
#> [29] "New Hampshire"        "New Jersey"
#> [31] "New Mexico"           "New York"
#> [33] "North Carolina"       "North Dakota"
#> [35] "Ohio"                 "Oklahoma"
#> [37] "Oregon"               "Pennsylvania"
#> [39] "Rhode Island"         "South Carolina"
#> [41] "South Dakota"         "Tennessee"
#> [43] "Texas"                "Utah"
#> [45] "Vermont"              "Virginia"
#> [47] "Washington"           "West Virginia"
#> [49] "Wisconsin"            "Wyoming"
#> [51] "District of Columbia"
```

Rather than use `render()` with the `input` and `output_file` arguments, as you did earlier, you can pass it the `params` argument to give it parameters to use when knitting. To do so, create a tibble with the information needed to render all 51 reports and save it as an object called `reports`, which you'll pass to the `render()` function, as follows:

```
reports <- tibble(
  input = "urban-covid-budget-report.Rmd",
  output_file = str_glue("{state}.xhtml"),
  params = map(state, ~ list(state = .))
)
```

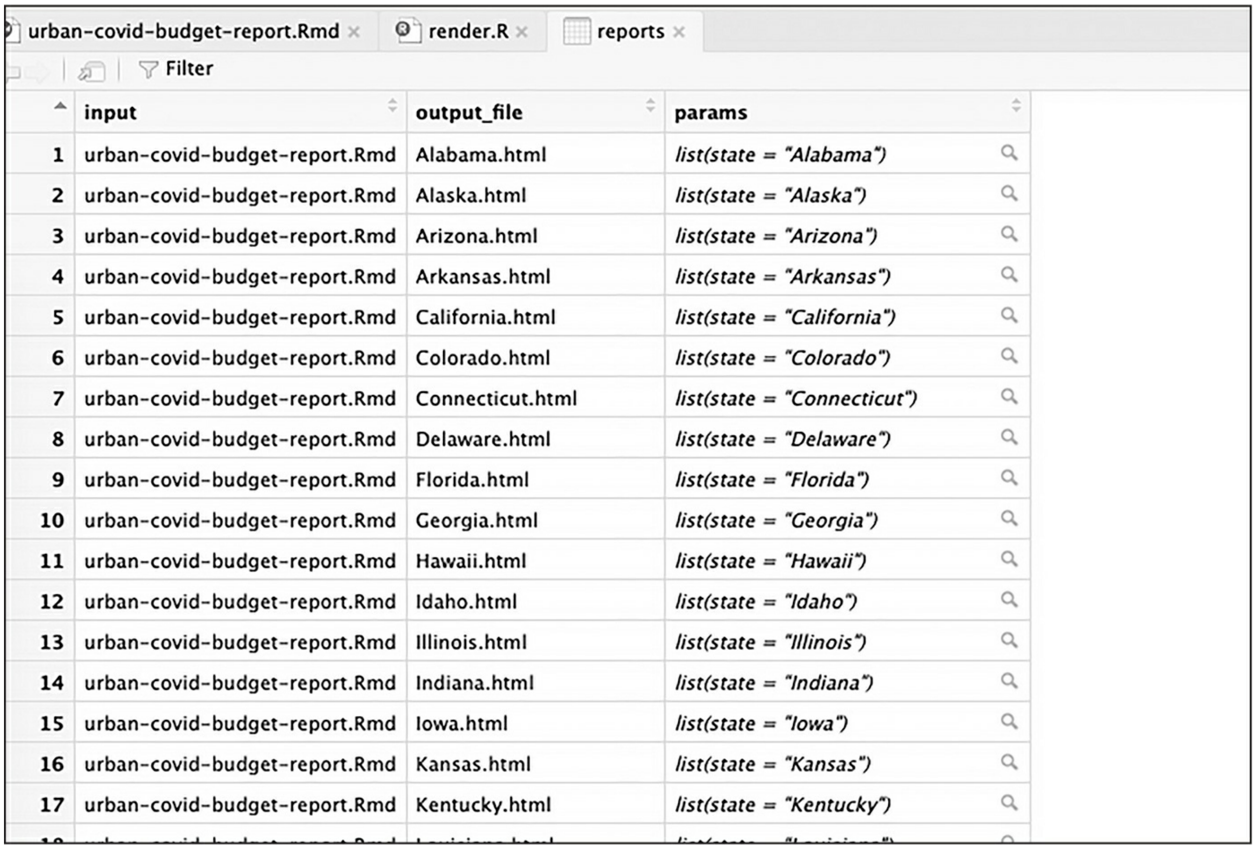
This code generates a tibble with 51 rows and 3 variables. In all rows, the `input` variable is set to the name of the R Markdown document. The value of `output_file` is set with `str_glue()` to be equal to the name of the state, followed by.*html* (for example, *Alabama.xhtml*).

The `params` variable is the most complicated of the three. It is what's known as a *named list*. This data structure puts the data in the `state`: *`state_name`* format needed for the R Markdown document's YAML. The `map()` function from the `purrr` package creates the named list, telling R to set the value of each row as `state = "Alabama"`, then `state = "Alaska"`, and so on, for all of the states. You can see these variables in the `reports` tibble:

```
#> # A tibble: 51 × 3
#>    input                         output_file    params
#>    <chr>                         <glue>         <list>
#>  1 urban-covid-budget-report.Rmd Alabama.xhtml   <named li
st>
#>  2 urban-covid-budget-report.Rmd Alaska.xhtml    <named li
st>
#>  3 urban-covid-budget-report.Rmd Arizona.xhtml   <named li
st>
#>  4 urban-covid-budget-report.Rmd Arkansas.xhtml  <named li
st>
#>  5 urban-covid-budget-report.Rmd California...  <named lis
t>
```

```
#>   6 urban-covid-budget-report.Rmd Colorado.xhtml  <named li
st>
#>   7 urban-covid-budget-report.Rmd Connecticut... <named lis
t>
#>   8 urban-covid-budget-report.Rmd Delaware.xhtml  <named li
st>
#>   9 urban-covid-budget-report.Rmd Florida.xhtml   <named li
st>
#> 10 urban-covid-budget-report.Rmd Georgia.xhtml   <named li
st>
#> # ... with 41 more rows
```

The `params` variable shows up as `<named list>`, but if you open the tibble in the RStudio viewer (click **reports** in the Environment tab), you can see the output more clearly, as shown in Figure 7-3.



	input	output_file	params
1	urban-covid-budget-report.Rmd	Alabama.html	list(state = "Alabama")
2	urban-covid-budget-report.Rmd	Alaska.html	list(state = "Alaska")
3	urban-covid-budget-report.Rmd	Arizona.html	list(state = "Arizona")
4	urban-covid-budget-report.Rmd	Arkansas.html	list(state = "Arkansas")
5	urban-covid-budget-report.Rmd	California.html	list(state = "California")
6	urban-covid-budget-report.Rmd	Colorado.html	list(state = "Colorado")
7	urban-covid-budget-report.Rmd	Connecticut.html	list(state = "Connecticut")
8	urban-covid-budget-report.Rmd	Delaware.html	list(state = "Delaware")
9	urban-covid-budget-report.Rmd	Florida.html	list(state = "Florida")
10	urban-covid-budget-report.Rmd	Georgia.html	list(state = "Georgia")
11	urban-covid-budget-report.Rmd	Hawaii.html	list(state = "Hawaii")
12	urban-covid-budget-report.Rmd	Idaho.html	list(state = "Idaho")
13	urban-covid-budget-report.Rmd	Illinois.html	list(state = "Illinois")
14	urban-covid-budget-report.Rmd	Indiana.html	list(state = "Indiana")
15	urban-covid-budget-report.Rmd	Iowa.html	list(state = "Iowa")
16	urban-covid-budget-report.Rmd	Kansas.html	list(state = "Kansas")
17	urban-covid-budget-report.Rmd	Kentucky.html	list(state = "Kentucky")

Figure 7-3: The named list column in the RStudio viewer

This view allows you to see the named list in the `params` variable, with the `state` variable equal to the name of each state.

Once you've created the `reports` tibble, you're ready to render the reports. The code to do so is only one line long:

```
pwalk(reports, render)
```

This `pwalk()` function (from the `purrr` package) has two arguments: a data frame or tibble (`reports`, in this case) and a function that runs for each row of this tibble, `render()`.

NOTE

You don't include the open and closing parentheses when passing the `render()` function to `pwalk()`.

Running this code runs the `render()` function for each row in `reports`, passing in the values for `input`, `output_file`, and `params`. This is equivalent to entering code like the following to run the `render()` function 51 times (for 50 states plus the District of Columbia):

```
render(
  input = "urban-covid-budget-report.Rmd",
  output_file = "Alabama.xhtml",
  params = list(state = "Alabama")
)

render(
  input = "urban-covid-budget-report.Rmd",
  output_file = "Alaska.xhtml",
  params = list(state = "Alaska")
)

render(
  input = "urban-covid-budget-report.Rmd",
  output_file = "Arizona.xhtml",
  params = list(state = "Arizona")
)
```

Here's the full R script file:

```
# Load packages
library(tidyverse)
library(rmarkdown)

# Create a vector of all states and the District of Columbia
state <- tibble(state.name) %>%
  rbind("District of Columbia") %>%
  pull(state.name)

# Create a tibble with information on the:
# input R Markdown document
# output HTML file
# parameters needed to knit the document
reports <- tibble(
  input = "urban-covid-budget-report.Rmd",
  output_file = str_glue("{state}.xhtml"),
  params = map(state, ~ list(state = .))
)

# Generate all of our reports
pwalk(reports, render)
```

After running the `pwalk(reports, render)` code, you should see 51 HTML documents appear in the files pane in RStudio. Each document consists of a report for that state, complete with a customized graph and accompanying text.

Best Practices

While powerful, parameterized reporting can present some challenges. For example, make sure to consider outliers in your data. In the case of the state reports, Washington, DC, is an outlier because it isn’t technically a state. The Urban Institute team altered the language in the report text so that it didn’t refer to Washington, DC, as a state by using an `if_else()` statement, as you saw at the beginning of this chapter.

Another best practice is to manually generate and review the reports whose parameter values have the shortest (Iowa, Ohio, and Utah in the state fiscal briefs) and longest (District of Columbia) text lengths. This way, you can identify places where the text length may have unexpected results, such as cut-off chart titles or page breaks disrupted by text running onto multiple lines. A few minutes of manual review can make the process of autogenerating multiple reports much smoother.

Summary

In this chapter, you re-created the Urban Institute's state fiscal briefs using parameterized reporting. You learned how to add a parameter to your R Markdown document, then use an R script to set the value of that parameter and knit the report.

Automating report production can be a huge time-saver, especially as the number of reports you need to generate grows. Consider another project at the Urban Institute: making county-level reports. With over 3,000 counties in the United States, creating these reports by hand isn't realistic. Not only that, but if the Urban Institute employees were to make their reports using SPSS, Excel, and Word, they would have to copy and paste values between programs. Humans are fallible, and mistakes occur, no matter how hard we try to avoid them. Computers, on the other hand, never make copy-and-paste errors. Letting computers handle the tedious work of generating multiple reports reduces the chance of error significantly.

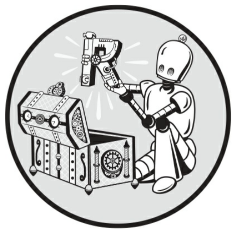
When you're starting out, parameterized reporting might feel like a heavy lift, as you have to make sure that your code works for every version of your report. But once you have your R Markdown document and accompanying R script file, you should find it easy to produce multiple reports at once, saving you work in the end.

Additional Resources

- Data@Urban Team, "Iterated Fact Sheets with R Markdown," Medium, July 24, 2018, *https://urban-institute.medium.com/iterated-fact-sheets-with-r-markdown-d685eb4eafce*.
- Data@Urban Team, "Using R Markdown to Track and Publish State Data," Medium, April 21, 2021, *https://urban-institute.medium.com/using-r-markdown-to-track-and-publish-state-data-d1291bfa1ec0*.

8

SLIDESHOW PRESENTATIONS



If you need to create a slideshow presentation, like one you might create in PowerPoint, R has you covered. In this chapter, you'll learn how to produce pre- sentations using `xaringan`. This package, which uses R Markdown, is the most widely used tool for creating slideshows in R.

You'll use `xaringan` to turn the penguin report from Chapter 6 into a slideshow. You'll learn how to create new slides, selectively reveal content, adjust text and image alignment, and style your presentation with CSS.

Why Use xaringan?

You might have noticed the Presentation option while creating a new R Markdown document in RStudio. This option offers several ways to make slides, such as knitting an R Markdown document to PowerPoint. However, using the `xaringan` package provides advantages over these options.

For example, because `xaringan` creates slides as HTML documents, you can post them online versus having to email them or print them out for viewers. You can send someone the presentation simply by sharing a link. Chapter 9 will discuss ways to publish your presentations online.

A second benefit of using `xaringan` is accessibility. HTML documents are easy to manipulate, giving viewers control over their appearance. For example, people with limited vision can access HTML documents in ways that allow them to view the content, such as by increasing the text size or using screen readers. Making presentations with `xaringan` lets more people engage with your slides.

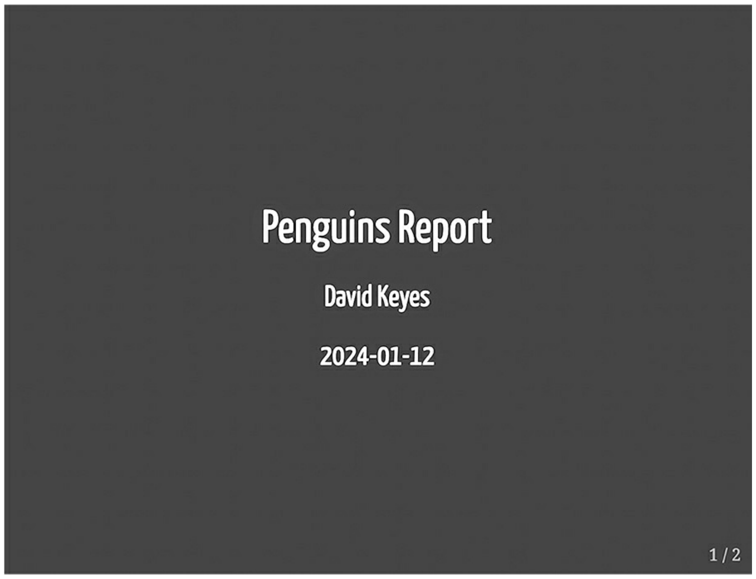
How xaringan Works

To get started with `xaringan`, run **`install.packages("xaringan")`** in RStudio to install the package. Next, navigate to **File** ▸ **New File** ▸ **R Markdown** to create a new project. Choose the **From Template** tab and select the template called **Ninja Presentation**, then click **OK**.

You should get an R Markdown document containing some default content. Delete this and add the penguin R report you created in Chapter 6. Then, change the output format in the YAML to `xaringan::moon_reader` like so:

```
title: "Penguins Report"
author: "David Keyes"
date: "2024-01-12"
output: xaringan::moon_reader
```

The `moon_reader` output format takes R Markdown documents and knits them as slides. Try clicking **Knit** to see what this looks like. You should get an HTML file with the same name as the R Markdown document (such as *xaringan-example.xhtml*), as shown in Figure 8-1.


Penguins Report

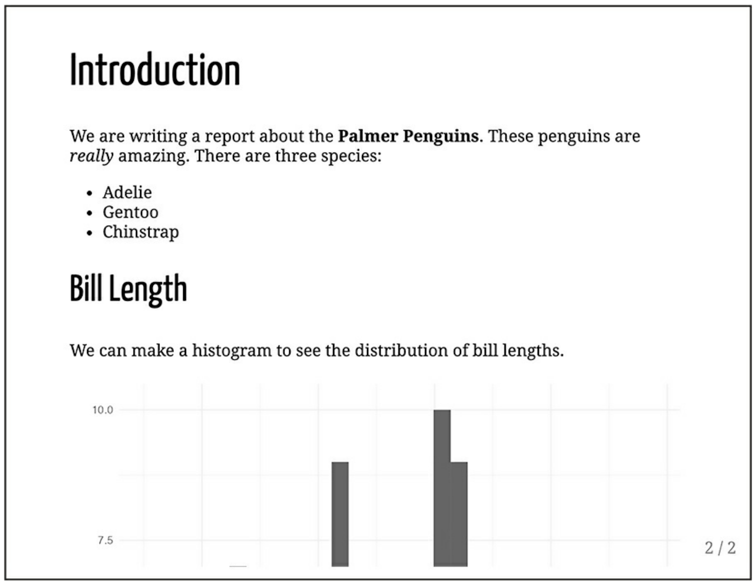
David Keyes

2024-01-12

1 / 2

Figure 8-1: The xaringan package automatically generates a title slide.

If you scroll to the next slide with the right arrow key, you should see familiar content. Figure 8-2 shows the second slide, which has the same text as the report from Chapter 6 and a cut-off version of its histogram.


Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

Bill Length

We can make a histogram to see the distribution of bill lengths.

Figure 8-2: The second slide needs adjustment, as the histogram is cut off.

Although the syntax for making slides with `xaringan` is nearly identical to that used to make reports with R Markdown, you need to make a few tweaks so that the content can fit on the slides. When you're working in a document that will be knitted to Word, its length doesn't matter, because reports can have 1 page or 100 pages. Working with `xaringan`, however, requires you to consider how much content can fit on a single slide. The cut-off histogram demonstrates what happens if you don't. You'll fix it next.

Creating a New Slide

You'll make this histogram fully visible by putting it in its own slide. To make a new slide, add three dashes (`---`) where you'd like it to begin. I've added them before the histogram code:

~~~
---

## Bill Length

We can make a histogram to see the distribution of bill lengt
hs.

```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

When you knit the document again, what was one slide should now be broken into two: an Introduction slide and a Bill Length slide. However, if you look closely, you'll notice that the bottom of the histogram is still slightly cut off. To correct this, you'll change its size.

Adjusting the Size of Figures

Adjust the size of the histogram using the code chunk option `fig.height`:

```
---

## Bill Length

We can make a histogram to see the distribution of bill lengt
hs.
```

~~~
```{r fig.height = 4}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

Doing this fits the histogram fully on the slide and also reveals the text that was hidden below it. Keep in mind that `fig.height` adjusts only the figure's output height; sometimes you may need to adjust the output width using `fig.width` in addition or instead.

Revealing Content Incrementally

When presenting a slideshow, you might want to show only a portion of the content on each slide at a time. Say, for example, that when you're presenting the first slide, you want to talk a bit about each penguin species. Rather than show all three species when you open this slide, you might prefer to have the names come up one at a time.

You can do this using a feature `xaringan` calls *incremental reveal*. Place two dashes (`--`) between any content you want to display incrementally, like so:

```
# Introduction

We are writing a report about the **Palmer Penguins**. These
penguins are
*really* amazing. There are three species:

- Adelie

--

- Gentoo

--

- Chinstrap
```

This code lets you show Adelie onscreen first; then Adelie and Gentoo;

and then Adelie, Gentoo, and Chinstrap.

When presenting your slides, use the right arrow to incrementally reveal the species.

Aligning Content with Content Classes

You'll also likely want to control how your content is aligned. To do so, you add the *content classes* `.left[]`, `.right[]`, and `.center[]` to specify the desired alignment for a piece of content. For example, to center-align the histogram, use `.center[]` as follows:

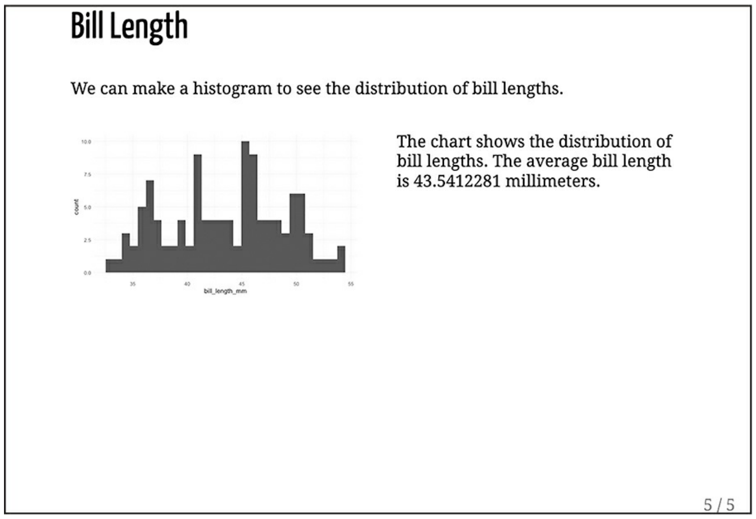
~~~
.center[
```{r fig.height = 4}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
]
~~~

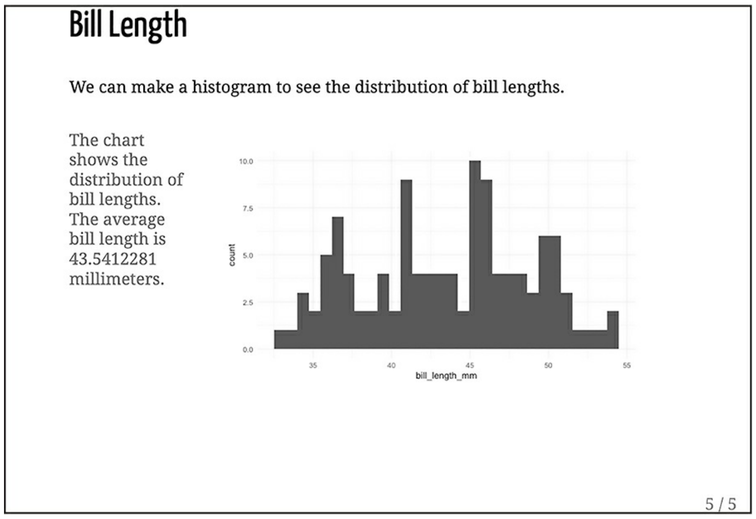
This code centers the chart on the slide.

Other built-in options can make two-column layouts. Adding `.pull-left[]` and `.pull-right[]` will make two equally spaced columns. Use the following code to display the histogram on the left side of the slide and the accompanying text on the right:

~~~
.pull-left[
```{r fig.height = 4}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
]

.pull-right[
```{r}
average_bill_length <- penguins %>%
  summarize(avg_bill_length = mean(bill_length_mm,
                                   na.rm = TRUE)) %>%
  pull(avg_bill_length)
~~~

```
```

The chart shows the distribution of bill lengths. The average
 bill length is
`r average_bill_length` millimeters.
]
```

Figure 8-3 shows the result.



Bill Length

We can make a histogram to see the distribution of bill lengths.

The chart shows the distribution of bill lengths. The average bill length is 43.5412281 millimeters.

Figure 8-3: A slide with two columns of equal size

To make a narrow left column and wide right column, use the content classes `.left-column[]` and `.right-column[]`. Figure 8-4 shows what the slide looks like with the text on the left and the histogram on the right.
```



## Bill Length

We can make a histogram to see the distribution of bill lengths.

The chart shows the distribution of bill lengths. The average bill length is 43.5412281 millimeters.

*Figure 8-4: A slide with a smaller left column and a larger right column*

In addition to aligning particular pieces of content on slides, you can also horizontally align the entire content using the `left`, `right`, and `center` classes. To do so, specify the class right after the three dashes that indicate a new slide, but before any content:

~~~
---

class: center

## Bill Length

We can make a histogram to see the distribution of bill lengt
hs.

```{r fig.height = 4}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
```
~~~

This code produces a horizontally centered slide. To adjust the vertical position, you can use the classes `top`, `middle`, and `bottom`.

Adding Background Images to Slides

Using the same syntax you just used to center the entire slide, you can also add a background image. Create a new slide, use the classes `center` and `middle` to horizontally and vertically align the content, and add a background image by specifying the path to the image within the parentheses of `url()`:

```
class: center, middle
background-image: url("penguins.jpg")

## Penguins
```

To run this code, you'll need a file called *penguins.jpg* in your project (you can download it at *https://data.rfortherestofus.com/penguins.jpg*). Knitting the document should produce a slide that uses this image as a background with the text *Penguins* in front of it, as shown in Figure 8-5.

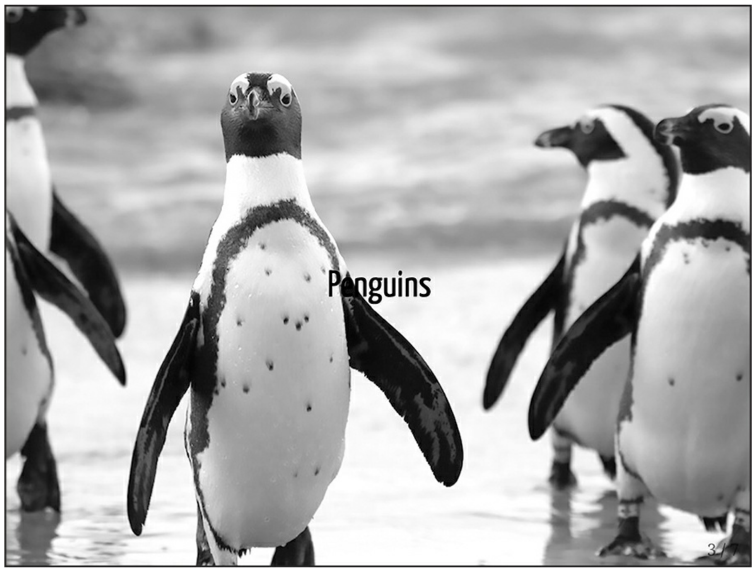


Figure 8-5: A slide that uses a background image

Now you'll add custom CSS to further improve this slide.

Applying CSS to Slides

One issue with the slide you just made is that the word *Penguins* is hard to read. It would be better if you could make the text bigger and a different color. To do this, you'll need to use *Cascading Style Sheets (CSS)*, the language used to style HTML documents. If you're thinking, *I'm reading this book to learn R, not CSS*, don't worry: you'll need only a bit of CSS to make tweaks to your slides. To apply them, you can write your own custom code, use a CSS theme, or combine the two approaches using the `xaringanthemer` package.

Custom CSS

To add custom CSS, create a new code chunk and place `css` between the curly brackets:

~~~
```{css}
.remark-slide-content h2 {
  font-size: 150px;
  color: white;
}
```
~~~

This code chunk tells R Markdown to make the second-level header (`h2`) 150 pixels large and white. Adding `.remark-slide-content` before the header targets specific elements in the presentation. The term *remark* comes from *remark.js*, a JavaScript library for making presentations that `xaringan` uses under the hood.

To change the font in addition to the text's size and color, add this CSS:

~~~
```{css}
@import url('https://fonts.googleapis.com/css2?family=Inter:w
ght@400;700&display=swap');

.remark-slide-content h2 {
  font-size: 150px;
  color: white;
  font-family: Inter;
  font-weight: bold;
}
```
~~~

The first new line makes a font called Inter available to the slides, because some people might not have the font installed on their computers. Next, this code applies Inter to the header and makes it bold. You can see the slide with bold Inter font in Figure 8-6.

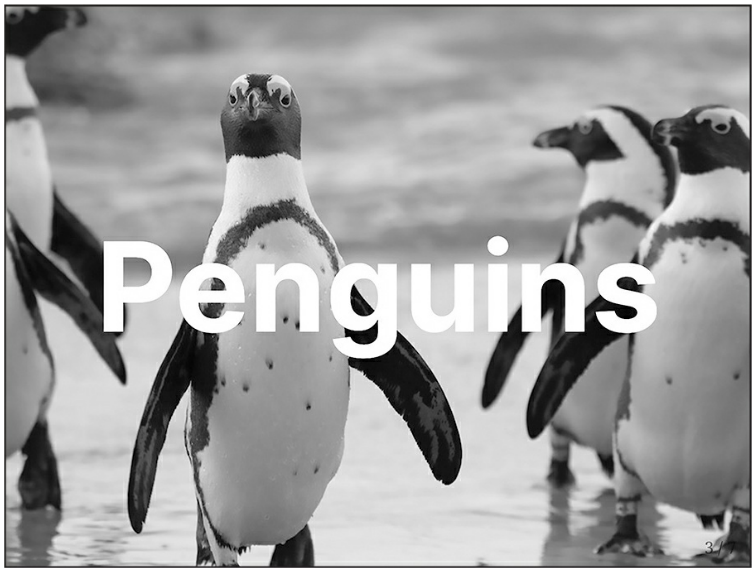


Figure 8-6: The title slide with CSS changes to the font

Because `xaringan` slides are built as HTML documents, you can customize them with CSS however you'd like. The sky's the limit!

Themes

You may not care to know the ins and outs of CSS. Fortunately, you can customize your slides in two ways without writing any CSS yourself. The first way is to apply `xaringan` themes created by other R users. Run this code to get a list of all available themes:

```
names(xaringan::list_css())
```

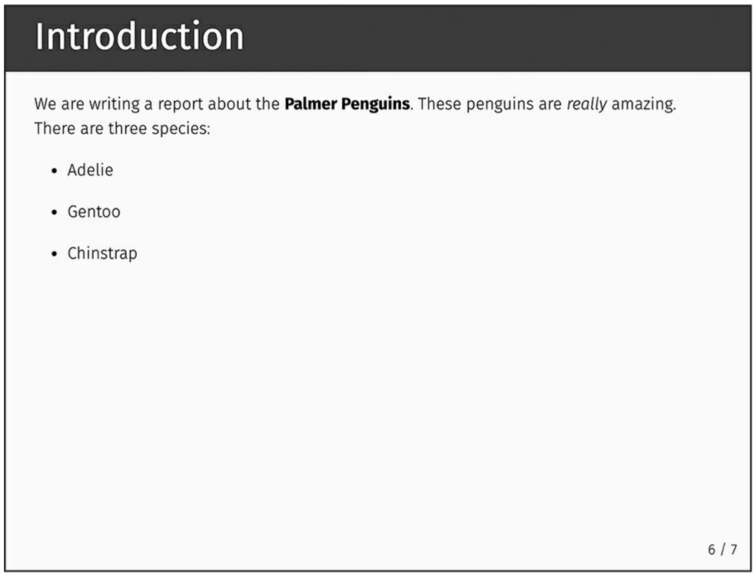
The output should look something like this:

```
#>  [1] "chocolate-fonts"  "chocolate"
#>  [3] "default-fonts"    "default"
#>  [5] "duke-blue"        "fc-fonts"
#>  [7] "fc"               "glasgow_template"
#>  [9] "hygge-duke"       "hygge"
#> [11] "ki-fonts"         "ki"
#> [13] "kunoichi"         "lucy-fonts"
#> [15] "lucy"                "metropolis-fonts"
#> [17] "metropolis"          "middlebury-fonts"
#> [19] "middlebury"          "nhsr-fonts"
#> [21] "nhsr"                "ninjutsu"
#> [23] "rladies-fonts"       "rladies"
#> [25] "robot-fonts"         "robot"
#> [27] "rutgers-fonts"       "rutgers"
#> [29] "shinobi"             "tamu-fonts"
#> [31] "tamu"                "uio-fonts"
#> [33] "uio"                 "uo-fonts"
#> [35] "uo"                  "uol-fonts"
#> [37] "uol"                 "useR-fonts"
#> [39] "useR"                "uwm-fonts"
#> [41] "uwm"                 "wic-fonts"
#> [43] "wic"
```

Some CSS files change fonts only, while others change general elements, such as text size, colors, and whether slide numbers are displayed. Using prebuilt themes usually requires you to use both a general theme and a fonts theme, as follows:

```
---
title: "Penguins Report"
author: "David Keyes"
date: "2024-01-12"
output:
  xaringan::moon_reader:
    css: [default, metropolis, metropolis-fonts]
---
```

This code tells `xaringan` to use the default CSS, as well as customizations made in the `metropolis` and `metropolis-fonts` CSS themes. These come bundled with `xaringan`, so you don’t need to install any additional packages to access them. Figure 8-7 shows how the theme changes the look and feel of the slides.



Introduction

We are writing a report about the **Palmer Penguins**. These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

6 / 7

Figure 8-7: A slide using the metropolis theme

If writing custom CSS is the totally flexible but more challenging option for tweaking your `xaringan` slides, then using a custom theme is simpler but a lot less flexible. Custom themes allow you to easily use others' prebuilt CSS but not to tweak it further.

The xaringanthemer Package

A nice middle ground between writing custom CSS and applying someone else's theme is to use the `xaringanthemer` package by Garrick Aden-Buie. This package includes several built-in themes but also allows you to easily create your own custom theme. After installing the package, adjust the `css` line in your YAML to use the *xaringan-themer.css* file like so:

```
---
title: "Penguins Report"
author: "David Keyes"
date: "2024-01-12"
output:
  xaringan::moon_reader:
    css: xaringan-themer.css
---
```

Now you can customize your slides by using the `style_xaringan()` function. This function has over 60 arguments, enabling you to tweak nearly any part of your xaringan slides. To replicate the custom CSS you wrote earlier in this chapter using xaringanthemer, you'll use just a few of the arguments:

~~~
```{r}
library(xaringanthemer)

style_xaringan(
  header_h2_font_size = "150px",
  header_color = "white",
  header_font_weight = "bold",
  header_font_family = "Inter"
)
```
~~~

This code sets the header size to 150 pixels and makes all the headers use the bold, white Inter font.

One particularly nice thing about the xaringanthemer package is that you can use any font available on Google Fonts by simply adding its name to header_font_family or another argument that sets font families (text_font_family and code_font_family are the other two, for styling body text and code, respectively). This means you won't have to include the line that makes the Inter font available.

Summary

In this chapter, you learned how to create presentations using the xaringan package. You saw how to incrementally reveal content on slides, create multicolumn layouts, and add background images to slides. You also changed your slides' appearance by applying custom themes, writing your own CSS, and using the xaringanthemer package.

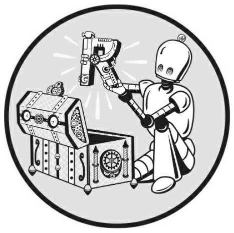
With xaringan, you can create any type of presentation you want and then customize it to match your desired look and feel. Creating presentations with xaringan also allows you to share your HTML slides easily and enables greater accessibility.

Additional Resources

- Garrick Aden-Buie, Silvia Canelón, and Shannon Pileggi, “Professional, Polished, Presentable: Making Great Slides with xaringan,” workshop materials, n.d., *https://presentable-user2021.netlify.app*.
- Silvia Canelón, “Sharing Your Work with xaringan: An Introduction to xaringan for Presentations: The Basics and Beyond,” workshop for the NHS-R Community 2020 Virtual Conference, November 2, 2020, *https://spcanelon.github.io/xaringan-basics-and-beyond/index.xhtml*.
- Alison Hill, “Meet xaringan: Making Slides in R Markdown,” slideshow presentation, January 16, 2019, *https://arm.rbind.io/slides/xaringan.xhtml*.
- Yihui Xie, J. J. Allaire, and Garrett Grolemund, “xaringan Presentations,” in *R Markdown: The Definitive Guide* (Boca Raton, FL: CRC Press, 2019), *https://bookdown.org/yihui/rmarkdown/*.

9

WEBSITES



During the summer of 2020, Matt Herman's family moved from Brooklyn to Westchester County, New York. It was still early in the COVID-19 pandemic, and Herman was shocked that the county published little data about infection rates. Vaccines weren't yet available, and daily choices like whether to go to the park depended on access to good data.

At the time, Herman was deputy director of the ChildStat Data Unit in the Office of Research and Analytics at the New York City Administration for Children's Services. This mouthful of a title meant he was skilled at working with data, enabling him to create the COVID resource he needed: the Westchester COVID-19 Tracking website.

Built entirely in R, this website uses charts, maps, tables, and text to summarize the latest COVID data for Westchester County. The website is no longer updated daily, but you can still view it at *https://westchester-covid.mattherman.info*.

To make this website, Herman wrote a set of R Markdown files and strung them together with the `distill` package. This chapter explains the basics of the package by walking through the creation of a simple website. You'll learn how to produce different page layouts, navigation menus, and interactive graphics, then explore strategies for hosting your website.

Creating a New distill Project

A website is merely a collection of HTML files like the one you produced in Chapter 8 when you created a slideshow presentation. The `distill` package uses multiple R Markdown documents to create several HTML files, then connects them with a navigation menu and more.

To create a `distill` website, install the package using **`install.packages ("distill")`**. Then start a project in RStudio by navigating to **File ▸ New Project ▸ New Directory** and selecting **Distill Website** as the project type.

Specify the directory and subdirectory where your project will live on your computer, then give your website a title. Check the Configure for GitHub Pages option, which provides an easy way to post your website online (you'll learn how it works in "GitHub Hosting" on page 154). Select it if you'd like to use this deployment option.

The Project Files

You should now have a project with several files. In addition to the *covid-website.Rproj* file indicating that you're working in an RStudio project, you should have two R Markdown documents, a *_site.yml* file, and a *docs* folder, where the rendered HTML files will go. Let's take a look at these website files.

R Markdown Documents

Each R Markdown file represents a page of the website. By default, `distill` creates a home page (*index.Rmd*) and an About page (*about.Rmd*) containing placeholder content. If you wanted to generate additional pages, you would simply add new R Markdown files, then list them in the *_site.yml* file discussed in the next section.

If you open the *index.Rmd* file, you'll notice that the YAML contains two arguments, `description` and `site`, that didn't appear in the R Markdown documents from previous chapters:

```
---
title: "COVID Website"
description: |
  Welcome to the website. I hope you enjoy it!
site: distill::distill_website
---
```

The `description` argument specifies the text that should go below the title of each page, as shown in Figure 9-1.

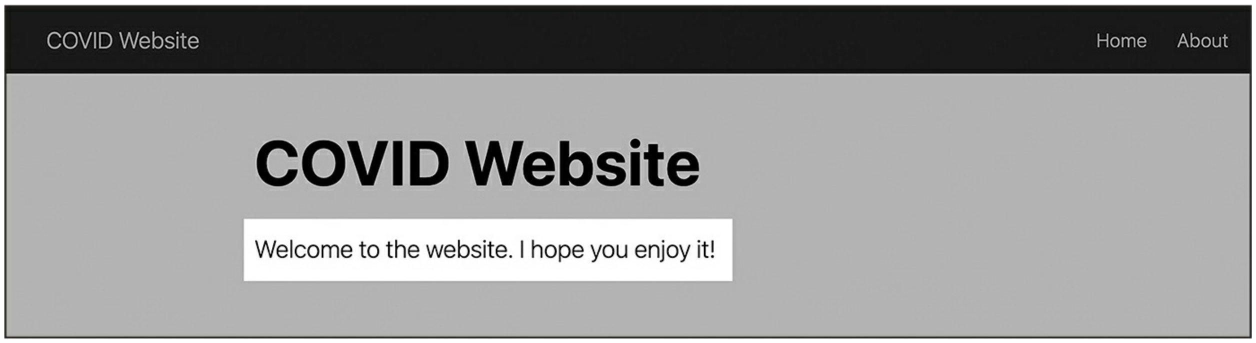


Figure 9-1: The default website description

The `site: distill::distill_website` line identifies the root page of a `distill` website. This means that when you knit the document, R Markdown knows to create a website rather than an individual HTML file and that the website should display this page first. The other pages of the website don't require this line. As long as they're listed in the *_site.yml* file, they'll be added to the site.

You'll also notice the absence of an argument you've seen in other R Markdown documents: `output`, which specifies the output format R should use while knitting. The reason `output` is missing here is that you'll specify the output for the entire website in the *_site.yml* file.

The _site.yml File

The *_site.yml* file tells R which R Markdown documents make up the website, what the knitted files should look like, what the website should be called, and more. When you open it, you should see the following code:

```
name: "covid-website"
title: "COVID Website"
description: |
  COVID Website
output_dir: "docs"
navbar:
  right:
    - text: "Home"
      href: index.xhtml
    - text: "About"
      href: about.xhtml
output: distill::distill_article
```

The `name` argument determines the URL for your website. By default, this should be the name of the directory where your `distill` project lives; in my case, that's the *covid-website* directory. The `title` argument creates the title for the entire website and shows up in the top left of the navigation bar by default. The `description` argument provides what's known as a *meta description*, which will show up as a couple of lines in Google search results to give users an overview of the website content.

The `output_dir` argument determines where the rendered HTML files live when you generate the website. You should see the *docs* directory listed here. However, you can change the output directory to any folder you choose.

Next, the `navbar` section defines the website's navigation. Here it appears on the right side of the header, but swapping the `right` parameter for `left` would switch its position. The navigation bar includes links to the site's two pages, Home and About, as shown in Figure 9-2.

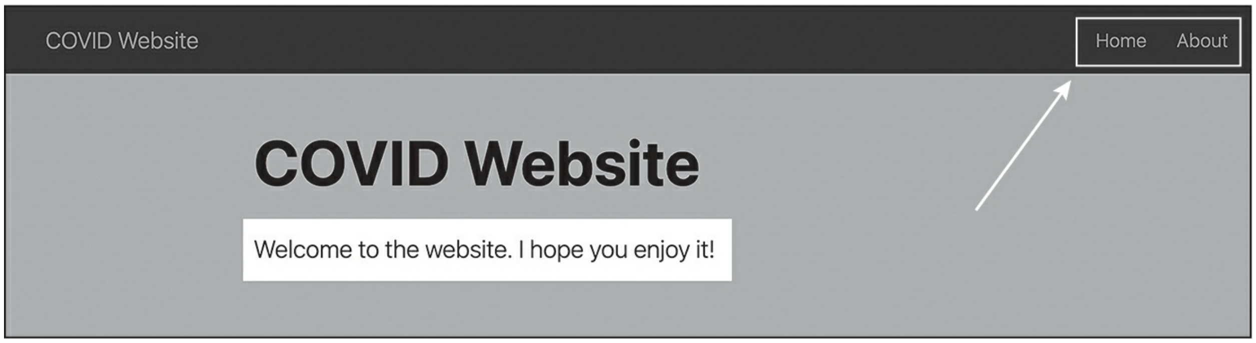


Figure 9-2: The website navigation bar

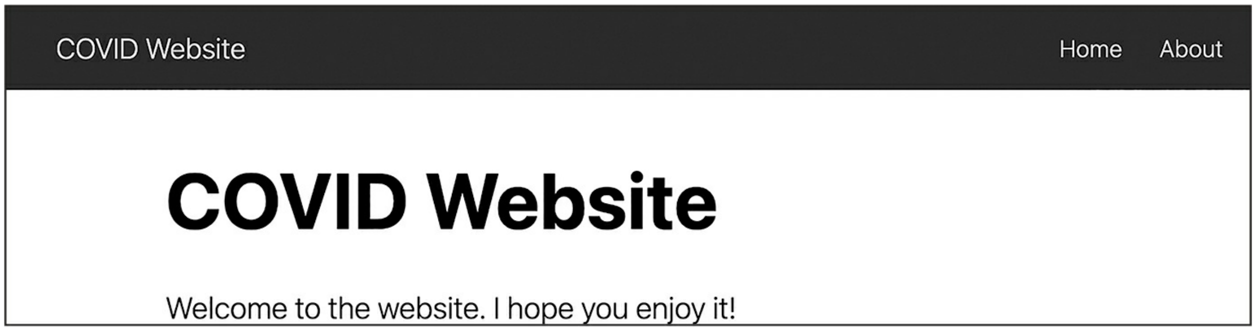
Within the `navbar` code, the `text` argument specifies what text shows up in the menu. (Try, for example, changing About to **About This Website**, and then change it back.) The `href` argument determines which HTML file the text in the navigation bar links to. If you want to include additional pages on your menu, you'll need to add both the `text` and `href` parameters.

Finally, the `output` argument specifies that all R Markdown documents should be rendered using the `distill_article` format. This format allows for layouts of different widths, *asides* (parenthetical items that live in a sidebar next to the main content), easily customizable CSS, and more.

Building the Site

We've explored the project's files but haven't yet used them to create the website. To do this, click **Build Website** in the Build tab of RStudio's top-right pane. (You could also run `rmarkdown::render_site()` in the console or in an R script file.)

This should render all R Markdown documents and add the top navigation bar to them with the options specified in the *_site.yml* file. To find the rendered files, look in *docs* (or whatever output directory you specified). Open the *index.xhtml* file and you'll find your website, which should look like Figure 9-3.



COVID Website

Welcome to the website. I hope you enjoy it!

Figure 9-3: The COVID website with default content

You can open any other HTML file as well to see its rendered version.

Applying Custom CSS

Websites made with `distill` tend to look similar, but you can change their design using custom CSS. The `distill` package even provides a function to simplify this process. Run **`distill::create_theme()`** in the console to create a file called *theme.css*, shown here:

```
/* base variables */

/* Edit the CSS properties in this file to create a custom
   Distill theme. Only edit values in the right column
   for each row; values shown are the CSS defaults.
   To return any property to the default,
   you may set its value to: unset
   All rows must end with a semi-colon.
*/

/* Optional: embed custom fonts here with `@import`
*/
/* This must remain at the top of this file.
*/

html {
  /*-- Main font sizes --*/
  --title-size:      50px;
  --body-size:       1.06rem;
  --code-size:       14px;
  --aside-size:      12px;
  --fig-cap-size:    13px;
  /*-- Main font colors --*/
  --title-color:     #000000;
  --header-color:    rgba(0, 0, 0, 0.8);
  --body-color:      rgba(0, 0, 0, 0.8);
  --aside-color:     rgba(0, 0, 0, 0.6);
  --fig-cap-color:   rgba(0, 0, 0, 0.6);
  /*-- Specify custom fonts ~~~ must be imported above --*/
  --heading-font:    sans-serif;
  --mono-font:       monospace;
  --body-font:       sans-serif;
  --navbar-font:     sans-serif;  /* websites + blogs only */
}

/*-- ARTICLE METADATA --*/
d-byline {
  --heading-size:    0.6rem;
  --heading-color:   rgba(0, 0, 0, 0.5);
  --body-size:       0.8rem;
```

```
  --body-color:      rgba(0, 0, 0, 0.8);
}

/*-- ARTICLE TABLE OF CONTENTS --*/
.d-contents {
  --heading-size:    18px;
  --contents-size:   13px;
}

/*-- ARTICLE APPENDIX --*/
d-appendix {
  --heading-size:    15px;
  --heading-color:   rgba(0, 0, 0, 0.65);
  --text-size:       0.8em;
  --text-color:      rgba(0, 0, 0, 0.5);
}

/*-- WEBSITE HEADER + FOOTER --*/
/* These properties only apply to Distill sites and blogs  */

.distill-site-header {
  --title-size:       18px;
  --text-color:       rgba(255, 255, 255, 0.8);
  --text-size:        15px;
  --hover-color:      white;
  --bkgd-color:       #0F2E3D;
}

.distill-site-footer {
  --text-color:       rgba(255, 255, 255, 0.8);
  --text-size:        15px;
  --hover-color:      white;
  --bkgd-color:       #0F2E3D;
}

/*-- Additional custom styles --*/
/* Add any additional CSS rules below                      */
```

Within this file is a set of CSS variables that allow you to customize the design of your website. Most of them have names that clearly show their purpose, and you can alter their default values to whatever you'd like. For example, the following edits to the site's header make the title and text size larger and change the background color to a light blue:

```
.distill-site-header {
  --title-size:       28px;
  --text-color:       rgba(255, 255, 255, 0.8);
  --text-size:        20px;
  --hover-color:      white;
  --bkgd-color:       #6cabdd;
}
```

Before you can see these changes, however, you need to add a line to the *_site.yml* file to tell `distill` to use this custom CSS when rendering:

```
name: "covid-website"
title: "COVID Website"
description: |
  COVID Website
theme: theme.css
output_dir: "docs"
navbar:
--snip--
```

Now you can generate the site again, and you should see your changes reflected.

There are a lot of other CSS variables in *theme.css* that you can change to tweak the appearance of your website. Playing around with them and regenerating your site is a great way to figure out what each one does.

NOTE

To learn more about customizing the look and feel of your website, check out the `distill` *websites made by others at* https://distillery.rbind.io.

Working with Website Content

You can add content to a page on your website by creating Markdown text and code chunks in the page's R Markdown document. For example, to highlight rates of COVID cases over time, you'll replace the contents of *index.Rmd* with code that displays a table, a map, and a chart on the website's home page. Here's the start of the file:

~~~
---
title: "COVID Website"
description: "Information about COVID rates in the United Sta
tes over time"
site: distill::distill_website
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE,
                      warning = FALSE,
                      message = FALSE)
~~~

```{r}
# Load packages

library(tidyverse)
library(janitor)
library(tigris)
library(gt)
library(lubridate)
```
```

After the YAML and `setup` code chunk, this code loads several packages, most of which you've seen in previous chapters: the `tidyverse` for data import, manipulation, and plotting (with ggplot); `janitor` for its `clean_names()` function, which makes the variable names easier to work with; `tigris` to import geospatial data about states; `gt` for making nice tables; and `lubridate` to work with dates.

Next, to import and clean the data, add this new code chunk:

```
```{r}
# Import data

us_states <- states(
  cb = TRUE,
  resolution = "20m",
  progress_bar = FALSE
) %>%
  shift_geometry() %>%
  clean_names() %>%
  select(geoid, name) %>%
```

```
  rename(state = name) %>%
  filter(state %in% state.name)

covid_data <- read_csv("https://raw.githubusercontent.com/nyt
imes/covid-19-data/master/rolling-averages/us-states.csv") %>
%
  filter(state %in% state.name) %>%
  mutate(geoid = str_remove(geoid, "USA-"))

last_day <- covid_data %>% ❶
  slice_max(
    order_by = date,
    n = 1
  ) %>%
  distinct(date) %>%
  mutate(date_nice_format = str_glue("{month(date, label = TR
UE, abbr = FALSE)} {day(date)},   {year(date)}")) %>%
  pull(date_nice_format)
```
```
# COVID Death Rates as of `r last_day`❷
```

This code uses the `slice_max()` function to get the latest date in the `covid_data` data frame. (Data was added until March 23, 2023, so that date is the most recent one.) From there, it uses `distinct()` to get a single observation of the most recent date (each date shows up multiple times in the `covid_data` data frame). The code then creates a `date_nice_format` variable using the `str_glue()` function to combine easy-to-read versions of the month, day, and year. Finally, the `pull()` function turns the data frame into a single variable called `last_day` ❶, which is referenced later in a text section. Using inline R code, this header now displays the current date ❷.

Include the following code to make a table showing the death rates per 100,000 people in four states (using all states would create too large a table):

~~~
```{r}
covid_data %>%
  filter(state %in% c(
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas"
~~~

```
  )) %>%
  slice_max(
    order_by = date,
    n = 1
  ) %>%
  select(state, deaths_avg_per_100k) %>%
  arrange(state) %>%
  set_names("State", "Death rate") %>%
  gt() %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_labels()
  )
```
```

This table resembles the code you saw in Chapter 5. First, the `filter()` function filters the data down to four states, and the `slice_max()` function gets the latest date. The code then selects the relevant variables (`state` and `deaths_avg_per_100k`), arranges the data in alphabetical order by state, sets the variable names, and pipes this output into a table made with the `gt` package.

Add the following code, which uses techniques covered in Chapter 4, to make a map of this data for all states:

````
We can see this same death rate data for all states on a map.

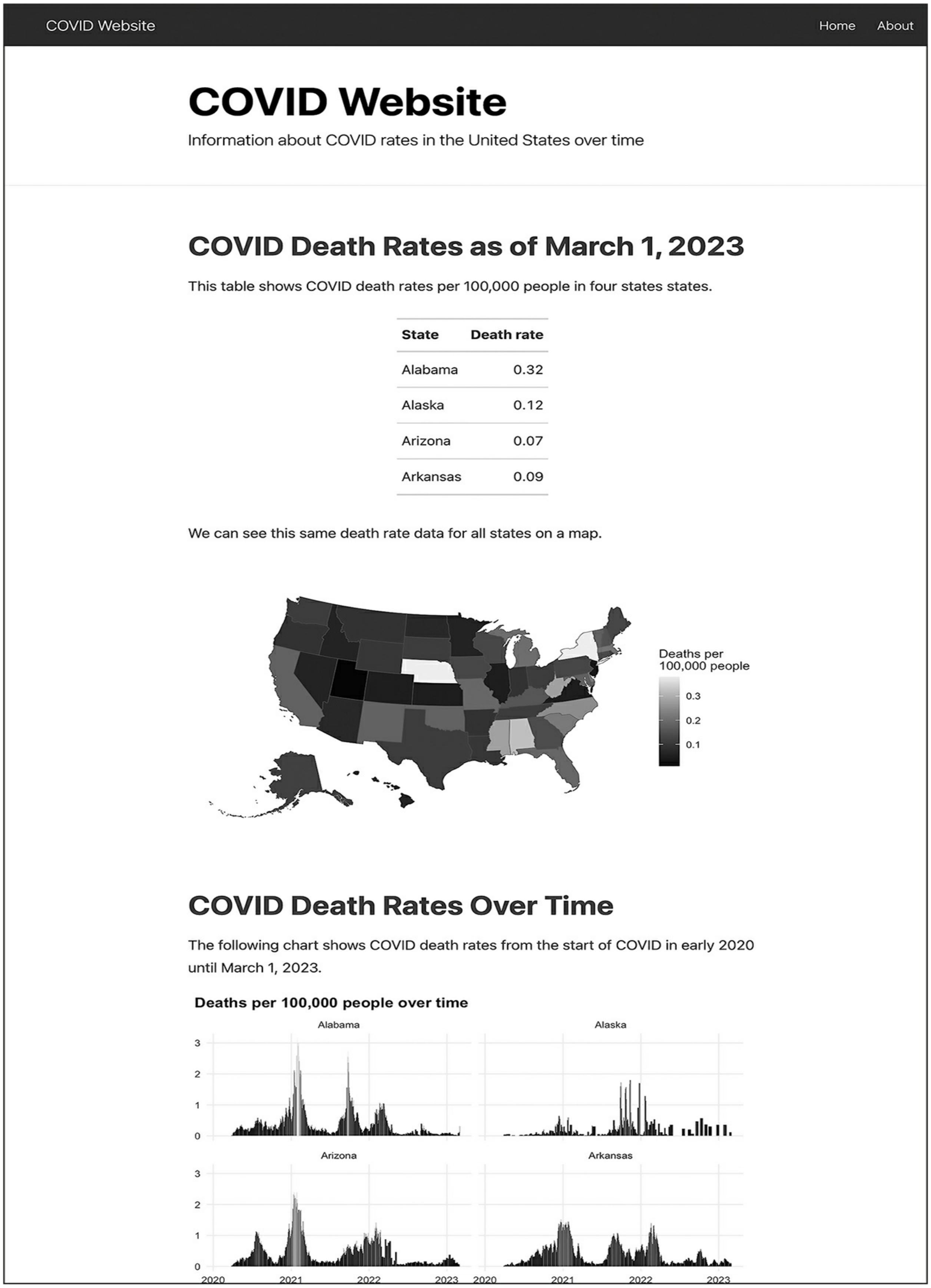
```{r}
most_recent <- us_states %>%
  left_join(covid_data, by = "state") %>%
  slice_max(order_by = date,
            n = 1)

most_recent %>%
  ggplot(aes(fill = deaths_avg_per_100k)) +
  geom_sf() +
  scale_fill_viridis_c(option = "rocket") +
  labs(fill = "Deaths per\n100,000 people") +
  theme_void()
```
````

This code creates a `most_recent` data frame by joining the `us_states`
````

geospatial data with the `covid_data` data frame before filtering to include only the most recent date. Then, it uses `most_recent` to create a map that shows deaths per 100,000 people.

Finally, to make a chart that shows COVID death rates over time in the four states from the table, add the following:

~~~
# COVID Death Rates Over Time

The following chart shows COVID death rates from the start of
 COVID in early
2020 until `r last_day`.

```{r}
covid_data %>%
  filter(state %in% c(
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas"
  )) %>%
  ggplot(aes(
    x = date,
    y = deaths_avg_per_100k,
    group = state,
    fill = deaths_avg_per_100k
  )) +
  geom_col() +
  scale_fill_viridis_c(option = "rocket") +
  theme_minimal() +
  labs(title = "Deaths per 100,000 people over time") +
  theme(
    legend.position = "none",
    plot.title.position = "plot",
    plot.title = element_text(face = "bold"),
    panel.grid.minor = element_blank(),
    axis.title = element_blank()
  ) +
  facet_wrap(
    ~state,
    nrow = 2
  )
```
~~~

Using the `geom_col()` function, this code creates a faceted set of bar charts that show change over time by state (faceting was discussed in Chapter 2). Finally, it applies the `rocket` color palette, applies `theme_minimal()`, and makes a few tweaks to that theme. Figure 9-4 shows what the website's home page looks like three years after the start of the pandemic.



COVID Website        Home    About

# COVID Website

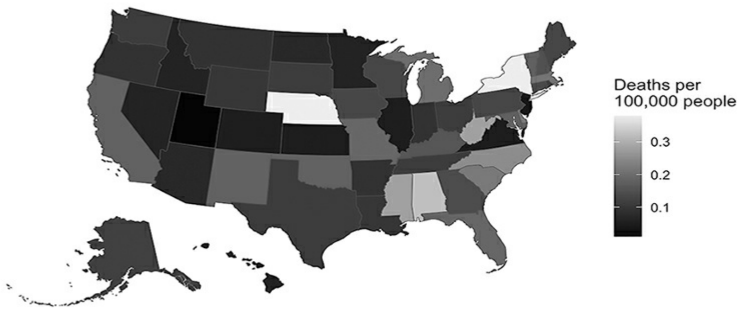
Information about COVID rates in the United States over time

## COVID Death Rates as of March 1, 2023

This table shows COVID death rates per 100,000 people in four states states.

| State | Death rate |
|---|---|
| Alabama | 0.32 |
| Alaska | 0.12 |
| Arizona | 0.07 |
| Arkansas | 0.09 |

We can see this same death rate data for all states on a map.



## COVID Death Rates Over Time

The following chart shows COVID death rates from the start of COVID in early 2020 until March 1, 2023.

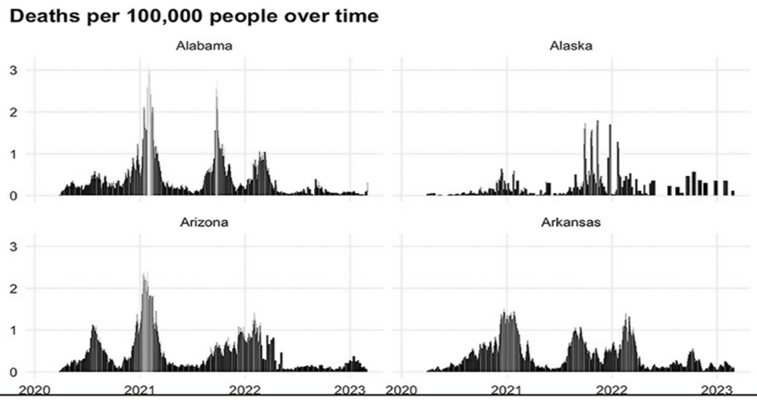


*Figure 9-4: The COVID website with a table, map, and chart*

Now that you have some content in place, you can tweak it. For example, because many states are quite small, especially in the Northeast, it's a bit challenging to see them. Let's look at how to make the entire map bigger.

## *Applying distill Layouts*

One nice feature of `distill` is that it includes four layouts you can apply to a code chunk to widen its output: `l-body-outset` (creates output that is a bit wider than the default), `l-page` (creates output that is wider still), `l-screen` (creates full-screen output), and `l-screen-inset` (creates full-screen output with a bit of a buffer).

Apply `l-screen-inset` to the map by modifying the first line of its code chunk as follows:

~~~
```{r layout = "l-screen-inset"}
~~~

This makes the map wider and taller and, as a result, much easier to read.

## *Making the Content Interactive*

The content you've added to the website so far is all static; it has none of the interactivity typically seen in websites, which often use JavaScript to respond to user behavior. If you're not proficient with HTML and JavaScript, you can use R packages like `distill`, `plotly`, and `DT`, which wrap JavaScript libraries, to add interactive elements like the graphics and maps Matt Herman uses on his Westchester County COVID website. Figure 9-5, for example, shows a tooltip that allows the user to see results for any single day.

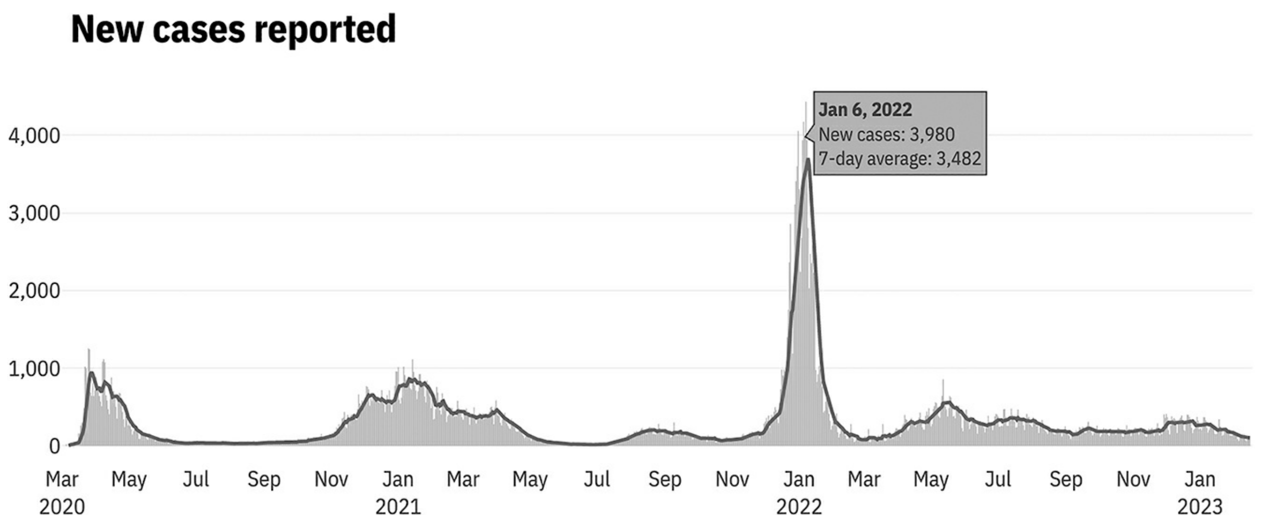
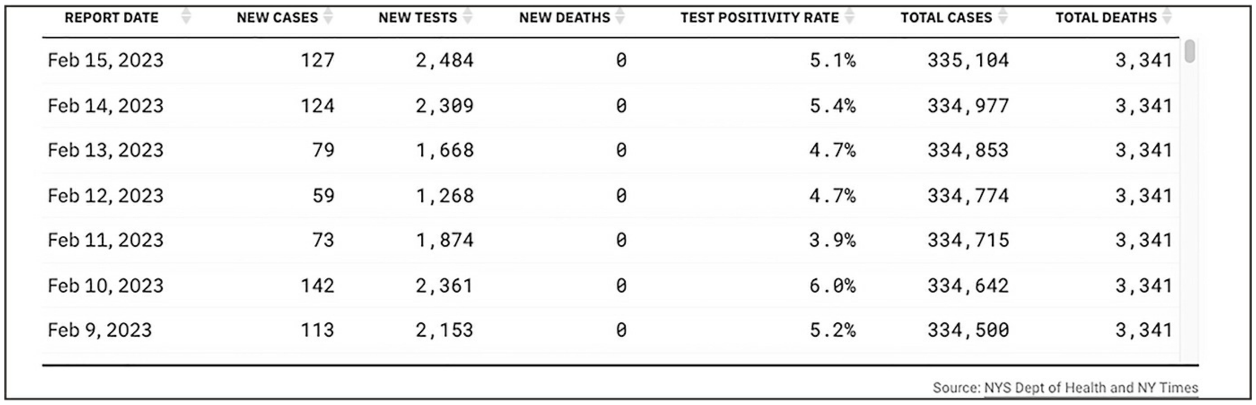


*Figure 9-5: An interactive tooltip showing new COVID cases by day*

Using the `DT` package, Herman also makes interactive tables that allow the user to scroll through the data and sort the values by clicking any variable in the header, as shown in Figure 9-6.



| REPORT DATE | NEW CASES | NEW TESTS | NEW DEATHS | TEST POSITIVITY RATE | TOTAL CASES | TOTAL DEATHS |
|---|---|---|---|---|---|---|
| Feb 15, 2023 | 127 | 2,484 | 0 | 5.1% | 335,104 | 3,341 |
| Feb 14, 2023 | 124 | 2,309 | 0 | 5.4% | 334,977 | 3,341 |
| Feb 13, 2023 | 79 | 1,668 | 0 | 4.7% | 334,853 | 3,341 |
| Feb 12, 2023 | 59 | 1,268 | 0 | 4.7% | 334,774 | 3,341 |
| Feb 11, 2023 | 73 | 1,874 | 0 | 3.9% | 334,715 | 3,341 |
| Feb 10, 2023 | 142 | 2,361 | 0 | 6.0% | 334,642 | 3,341 |
| Feb 9, 2023 | 113 | 2,153 | 0 | 5.2% | 334,500 | 3,341 |

Source: NYS Dept of Health and NY Times

*Figure 9-6: An interactive table made with the DT package*

Next, you'll add some interactivity to your COVID website, beginning with your table.

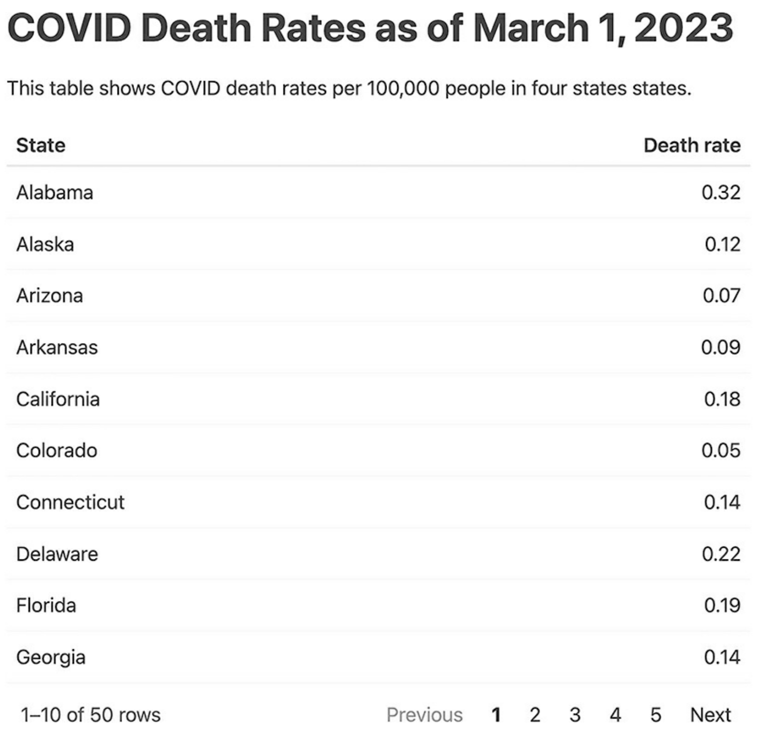
## Adding Pagination to a Table with reactable

Remember how you included only four states in the table to keep it from getting too long? By creating an interactive table, you can avoid this limitation. The `reactable` package is a great option for interactive tables.

First, install it with **`install.packages("reactable")`**. Then, swap out the `gt` package code you used to make your static table with the `reactable()` function to show all states:

```
library(reactable)

covid_data %>%
  slice_max(
    order_by = date,
    n = 1
  ) %>%
  select(state, deaths_avg_per_100k) %>%
  arrange(state) %>%
  set_names("State", "Death rate") %>%
  reactable()
```

The `reactable` package shows 10 rows by default and adds pagination, as shown in Figure 9-7.



COVID Death Rates as of March 1, 2023

This table shows COVID death rates per 100,000 people in four states states.

| State | Death rate |
|---|---|
| Alabama | 0.32 |
| Alaska | 0.12 |
| Arizona | 0.07 |
| Arkansas | 0.09 |
| California | 0.18 |
| Colorado | 0.05 |
| Connecticut | 0.14 |
| Delaware | 0.22 |
| Florida | 0.19 |
| Georgia | 0.14 |

1–10 of 50 rows

*Figure 9-7: An interactive table built with reactable*

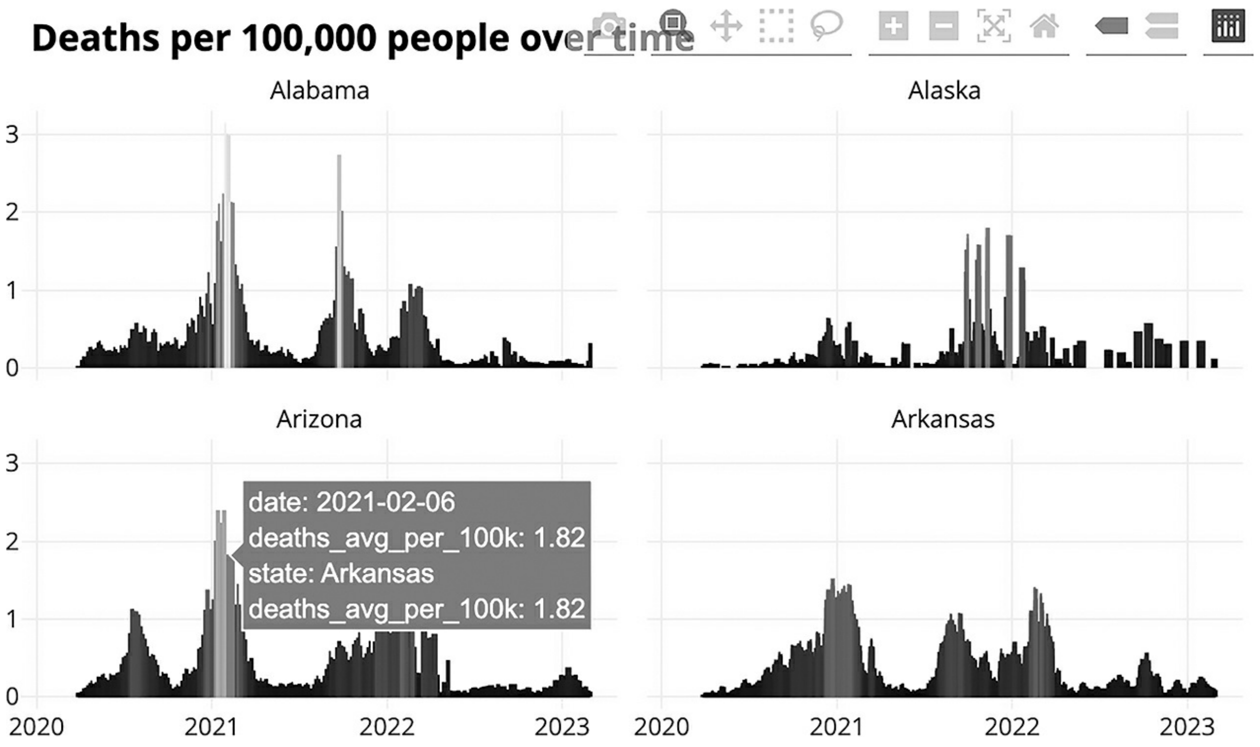
The `reactable()` function also enables sorting by default. Although you used the `arrange()` function in your code to sort the data by state name, users can click the "Death rate" column to sort values using that variable instead.

## Creating a Hovering Tooltip with plotly

Now you'll add some interactivity to the website's chart using the `plotly` package. First, install `plotly` with **`install.packages("plotly")`**. Then, create a plot with ggplot and save it as an object. Pass the object to the `ggplotly()` function, which turns it into an interactive plot, and run the following code to apply `plotly` to the chart of COVID death rates over time:

```
library(plotly)

covid_chart <- covid_data %>%
  filter(state %in% c(
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas"
  )) %>%
  ggplot(aes(
    x = date,
    y = deaths_avg_per_100k,
    group = state,
    fill = deaths_avg_per_100k
  )) +
  geom_col() +
  scale_fill_viridis_c(option = "rocket") +
  theme_minimal() +
  labs(title = "Deaths per 100,000 people over time") +
  theme(
    legend.position = "none",
    plot.title.position = "plot",
    plot.title = element_text(face = "bold"),
    panel.grid.minor = element_blank(),
    axis.title = element_blank()
  ) +
  facet_wrap(
    ~state,
    nrow = 2
  )

ggplotly(covid_chart)
```

This is identical to the chart code shown earlier in this chapter, except that now you're saving your chart as an object called `covid_chart` and then running `ggplotly(covid_chart)`. This code produces an interactive chart that shows the data for a particular day when a user mouses over it. But the tooltip that pops up, shown in Figure 9-8, is cluttered and overwhelming because the `ggplotly()` function shows all data by default.



```
date: 2021-02-06
deaths_avg_per_100k: 1.82
state: Arkansas
deaths_avg_per_100k: 1.82
```

*Figure 9-8: The plotly default produces a messy tooltip.*

To make the tooltip more informative, create a single variable containing the data you want to display and tell `ggplotly()` to use it:

```
covid_chart <- covid_data %>%
  filter(state %in% c(
    "Alabama",
    "Alaska",
    "Arizona",
    "Arkansas"
  )) %>%
  mutate(date_nice_format = str_glue("{month(date, label = TR
UE, abbr = FALSE)} {day(date)},   {year(date)}")) %>% ❶
```

```
  mutate(tooltip_text = str_glue("{state}<br>{date_nice_forma
t}<br>{deaths_avg_per_100k}   per 100,000 people")) %>% ❷
  ggplot(aes(
    x = date,
    y = deaths_avg_per_100k,
    group = state,
    text = tooltip_text,❸
    fill = deaths_avg_per_100k
  )) +
  geom_col() +
  scale_fill_viridis_c(option = "rocket") +
  theme_minimal() +
  labs(title = "Deaths per 100,000 people over time") +
  theme(
    legend.position = "none",
    plot.title.position = "plot",
    plot.title = element_text(face = "bold"),
    panel.grid.minor = element_blank(),
    axis.title = element_blank()
  ) +
  facet_wrap(
    ~state,
    nrow = 2
  )

ggplotly(
  covid_chart,
  tooltip = "tooltip_text"❹
)
```

This code begins by creating a `date_nice_format` variable that produces dates in the more readable format January 1, 2023, instead of 2023-01-01 ❶. This value is then combined with the state and death rate variables, and the result is saved as `tooltip_text` ❷. Next, the code adds a new aesthetic property in the `ggplot()` function ❸. This property doesn't do anything until it's passed to `ggplotly()`❹.

Figure 9-9 shows what the new tooltip looks like: it displays the name of the state, a nicely formatted date, and that day's death rate.

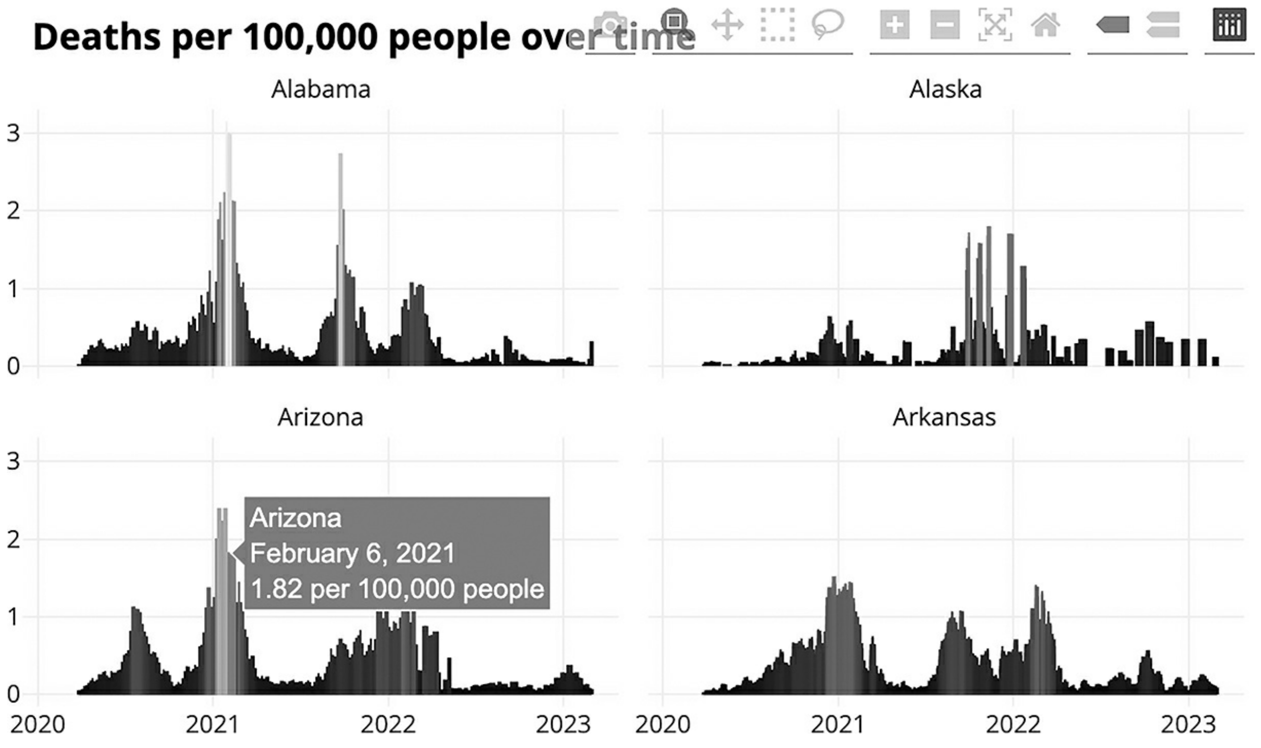


*Figure 9-9: Easy-to-read interactive tooltips on the COVID-19 death rate chart*

Adding interactivity is a great way to take advantage of the website medium. Users who might feel overwhelmed looking at the static chart can explore the interactive version, mousing over areas to see a summary of the results on any single day.

## Hosting the Website

Now that you've made a website, you need a way to share it. There are various ways to do this, ranging from simple to quite complex. The easiest solution is to compress the files in your *docs* folder (or whatever folder you put your rendered website in) and email your ZIP file to your recipients. They can unzip it and open the HTML files in their browser. This works fine if you know you won't want to make changes to your website's data or styles. But, as Chapter 5 discussed, most projects aren't really one-time events.

### *Cloud Hosting*

A better approach is to put your entire *docs* folder in a place where others can see it. This could be an internal network, Dropbox, Google Drive, Box, or something similar. Hosting the files in the cloud this way is simple to implement and allows you to control who can see your website.

You can even automate the process of copying your *docs* folder to various online file-sharing sites using R packages: the `rdrop2` package works with Dropbox, `googledrive` works with Google Drive, and `boxr` works with Box. For example, code like the following would automatically upload the project to Dropbox:

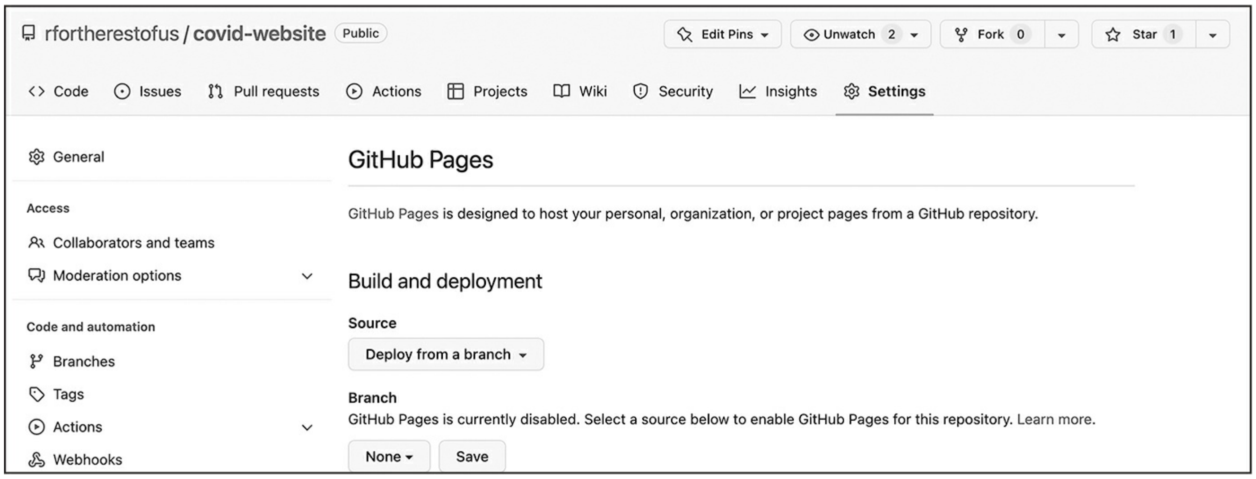
```
library(tidyverse)
library(rmarkdown)
library(fs)
library(rdrop2)

# Render the website
render_site()

# Upload to Dropbox
website_files <- dir_ls(
  path = "docs",
  type = "file",
  recurse = TRUE
)

walk(website_files, drop_upload, path = "COVID Website")
```

This code, which I typically add to a separate file called *render.R*, renders the site, uses the `dir_ls()` function from the `fs` package to identify all files in the *docs* directory, and then uploads these files to Dropbox. Now you can run your entire file to generate and upload your website in one go.

### GitHub Hosting

A more complicated yet powerful alternative to cloud hosting is to use a static hosting service like GitHub Pages. Each time you *commit* (take a snapshot of) your code and *push* (sync) it to GitHub, this service deploys the website to a URL you've set up. Learning to use GitHub is an investment of time and effort (the self-published book *Happy Git and GitHub for the useR* by Jenny Bryan at *https://happygitwithr.com* is a great resource), but being able to host your website for free makes it worthwhile.

Here's how GitHub Pages works. Most of the time, when you look at a file on GitHub, you see its underlying source code, so if you looked at an HTML file, you'd see only the HTML code. GitHub Pages, on the other hand, shows you the rendered HTML files. To host your website on GitHub Pages, you'll need to first push your code to GitHub. Once you have a repository set up there, go to it, then go to the **Settings** tab, which should look like Figure 9-10.



GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Build and deployment

Source

Deploy from a branch

Branch

GitHub Pages is currently disabled. Select a source below to enable GitHub Pages for this repository. Learn more.

None Save

*Figure 9-10: Setting up GitHub Pages*

Now choose how you want GitHub to deploy the raw HTML. The easiest approach is to keep the default source. To do so, select **Deploy** from a branch and then select your default branch (usually *main* or *master*). Next, select the directory containing the HTML files you want to be rendered. If you configured your website for GitHub Pages at the beginning of this chapter, the files should be in *docs*. Click **Save** and wait a few minutes, and GitHub should show the URL where your website now lives.

The best part about hosting your website on GitHub Pages is that any time you update your code or data, the website will update as well. R Markdown, `distill`, and GitHub Pages make building and maintaining websites a snap.

## Summary

In this chapter, you learned to use the `distill` package to make websites in R. This package provides a simple way to get a website up and running with the tool you’re already using for working with data. You’ve seen how to:

- Create new pages and add them to your top navigation bar
- Customize the look and feel of your website with tweaks to the CSS
- Use wider layouts to make content fit better on individual pages
- Convert static data visualization and tables into interactive versions
- Use GitHub Pages to host an always-up-to-date version of your website

Matt Herman has continued building websites with R. He and his colleagues at the Council of State Governments Justice Center have made a great website using Quarto, the language-agnostic version of R Markdown. This website, found at *https://projects.csgjusticecenter.org/tools-for-states-to-address-crime/*, highlights crime trends throughout the United States using many of the same techniques you saw in this chapter.

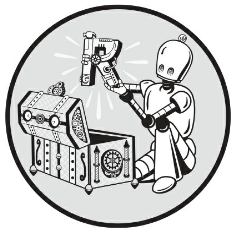
Whether you prefer `distill` or Quarto, using R is a quick way to develop complex websites without having to be a sophisticated frontend web developer. The websites look good and communicate well. They are one more example of how R can help you efficiently share your work with the world.

## Additional Resources

- The Distillery, “Welcome to the Distillery!,” accessed November 30, 2023, *https://distillery.rbind.io*.
- Thomas Mock, “Building a Blog with distill,” *The MockUp*, August 1, 2020, *https://themockup.blog/posts/2020-08-01-building-a-blog-with-distill/*.

# 10

# QUARTO



Quarto, the next-generation version of R Markdown, offers a few advantages over its predecessor. First, the syntax Quarto uses across output types is more consistent. As you've seen in this book, R Markdown documents might use a variety of conventions; for example, `xaringan` indicates new slides using three dashes, which would create a horizontal line in other output formats, and the `distill` package likewise has layout options that don't work in `xaringan`.

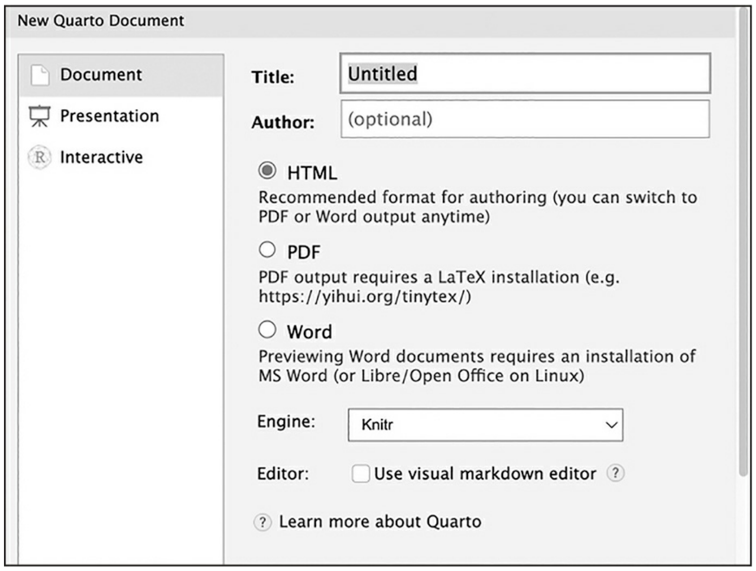
Quarto also supports more languages than R Markdown does, as well as multiple code editors. While R Markdown is designed to work specifically in the RStudio IDE, Quarto works not only in RStudio but also in code editors such as Visual Studio (VS) Code and JupyterLab, making it easy to use with multiple languages.

This chapter focuses on the benefits of using Quarto as an R user. It explains how to set up Quarto, then covers some of the most important differences between Quarto and R Markdown. Finally, you'll learn how to use Quarto to make the parameterized reports, presentations, and websites covered in previous chapters.

## Creating a Quarto Document

Versions of RStudio starting with 2022.07.1 come with Quarto installed. To check your RStudio version, click **RStudio ▸ About RStudio** in the top menu bar. If you have an older version of RStudio, update it now by reinstalling it, as outlined in Chapter 1. Quarto should then be installed for you.

Once you've installed Quarto, create a document by clicking **File ▸ New File ▸ Quarto Document**. You should see a menu, shown in Figure 10-1, that looks like the one used to create an R Markdown document.



New Quarto Document

Document
Presentation
Interactive

Title: Untitled
Author: (optional)

[✓ HTML]
Recommended format for authoring (you can switch to PDF or Word output anytime)

PDF
PDF output requires a LaTeX installation (e.g. https://yihui.org/tinytex/)

Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux)

Engine: Knitr

Editor: Use visual markdown editor

Learn more about Quarto

*Figure 10-1: The RStudio menu for creating a new Quarto document*

Give your document a title and choose an output format. The Engine option allows you to select a different way to render documents. By default, it uses Knitr, the same rendering tool used by R Markdown. The Use Visual Markdown Editor option provides an interface that looks more like Microsoft Word, but it can be finicky, so I won't cover it here.

The resulting Quarto document should contain default content, just as R Markdown documents do:

```
title: "My Report"
format: html
```

~~~
## Quarto

Quarto enables you to weave together content and executable c
ode into a
finished document. To learn more about Quarto see <https://qu
arto.org>.

## Running Code

When you click the **Render** button a document will be gener
ated that includes
both content and the output of embedded code. You can embed c
ode like this:

```{r}
1 + 1
~~~

You can add options to executable code like this:

```{r}
#| echo: false
2 * 2
```
The `echo: false` option disables the printing of code (only
output is displayed).
```

Although R Markdown and Quarto have many features in common, they also have some differences to be aware of.

## Comparing R Markdown and Quarto

Quarto and R Markdown documents have the same basic structure—YAML metadata, followed by a combination of Markdown text and code chunks—but they have some variations in syntax.

### *The format and execute YAML Fields*

Quarto uses slightly different options in its YAML. It replaces the `output` field with the `format` field and uses the value `html` instead of `html_document`:

```
---
title: "My Report"
```
```

```
format: html
---
```

Other Quarto formats also use different names than their R Markdown counterparts: `docx` instead of `word_document` and `pdf` instead of `pdf_document`, for example. All of the possible formats can be found at *https://quarto.org/docs/guide/*.

A second difference between R Markdown and Quarto syntax is that Quarto doesn't use a `setup` code chunk to set default options for showing code, charts, and other elements in the rendered versions of the document. In Quarto, these options are set in the `execute` field of the YAML. For example, the following would hide code, as well as all warnings and messages, from the rendered document:

```
---
title: "My Report"
format: html
execute:
  echo: false
  warning: false
  message: false
---
```

Quarto also allows you to write `true` and `false` in lowercase.

## *Individual Code Chunk Options*

In R Markdown, you override options at the individual code chunk level by adding the new option within the curly brackets that start a code chunk. For example, the following would show both the code `2 * 2` and its output:

~~~
```{r echo = TRUE}
2 * 2
```
~~~

Quarto instead uses this syntax to set individual code chunk–level options:

~~~
```{r}
~~~

~~~
#| echo: false
2 * 2
```
~~~

The option is set within the code chunk itself. The characters `#|` (known as a *hash pipe*) at the start of a line indicate that you are setting options.

### *Dashes in Option Names*

Another difference you're likely to see if you switch from R Markdown to Quarto is that option names consisting of two words are separated by a dash rather than a period. R Markdown, for example, uses the code chunk option `fig.height` to specify the height of plots. In contrast, Quarto uses `fig-height`, as follows:

~~~
```{r}
#| fig-height: 10

library(palmerpenguins)
library(tidyverse)

ggplot(
  penguins,
  aes(
    x = bill_length_mm,
    y = bill_depth_mm
  )
) +
  geom_point()
```
~~~

Helpfully for anyone coming from R Markdown, `fig.height` and similar options containing periods will continue to work if you forget to make the switch. A list of all code chunk options can be found on the Quarto website at *https://quarto.org/docs/reference/cells/cells-knitr.xhtml*.

### *The Render Button*

You can follow the same process to render your Quarto document as in R Markdown, but in Quarto the button is called Render rather than Knit. Clicking Render will turn the Quarto document into an HTML file, Word document, or any other output format you select.

## Parameterized Reporting

Now that you've learned a bit about how Quarto works, you'll make a few different documents with it, starting with a parameterized report. The process of making parameterized reports with Quarto is nearly identical to doing so with R Markdown. In fact, you can adapt the R Markdown document you used to make the Urban Institute COVID report in Chapter 7 for Quarto simply by copying the *.Rmd* file, changing its extension to *.qmd*, and then making a few other changes:

~~~
---
title: "Urban Institute COVID Report"
format: html❶
params:
  state: "Alabama"
execute:❷
  echo: false
  warning: false
  message: false
---

```{r}
library(tidyverse)
library(urbnthemes)
library(scales)
~~~

# `r params$state`

```{r}
cases <- tibble(state.name) %>%
  rbind(state.name = "District of Columbia") %>%
  left_join(
    read_csv("https://data.rfortherestofus.com/united_states_
covid19_cases_deaths
    _and_testing_by_state.csv", skip = 2),
    by = c("state.name" = "State/Territory")
  ) %>%
  select(
    total_cases = `Total Cases`,
```

```
    state.name,
    cases_per_100000 = `Case Rate per 100000`
  ) %>%
  mutate(cases_per_100000 = parse_number(cases_per_100000)) %
>%
  mutate(case_rank = rank(-cases_per_100000, ties.method = "m
in"))
```

```{r}
state_text <- if_else(params$state == "District of Columbia",
 str_glue("the District of
Columbia"), str_glue("state of {params$state}"))

state_cases_per_100000 <- cases %>%
  filter(state.name == params$state) %>%
  pull(cases_per_100000) %>%
  comma()

state_cases_rank <- cases %>%
  filter(state.name == params$state) %>%
  pull(case_rank)
```

In `r state_text`, there were `r state_cases_per_100000` case
s per 100,000 people in the last
seven days. This puts `r params$state` at number `r state_cas
es_rank` of 50 states and the
District of Columbia.

```{r}
#| fig-height: 8❸

set_urbn_defaults(style = "print")

cases %>%
  mutate(highlight_state = if_else(state.name == params$state
, "Y", "N")) %>%
  mutate(state.name = fct_reorder(state.name, cases_per_10000
0)) %>%
  ggplot(aes(
    x = cases_per_100000,
    y = state.name,
    fill = highlight_state
  )) +
  geom_col() +
  scale_x_continuous(labels = comma_format()) +
  theme(legend.position = "none") +
  labs(
    y = NULL,
    x = "Cases per 100,000"
  )
```
```

This code switches `output: html_document` to `format: html` in the YAML ❶, then removes the `setup` code chunk and sets those options in the YAML's `execute` field ❷. Finally, the `fig.height` option in the last code chunk is replaced with `fig-height` and labeled as an option with the hash pipe ❸.

Next, to create one report for each state, you must tweak the *render.R* script file you used to make parameterized reports in Chapter 7:

```
# Load packages
library(tidyverse)
❶ library(quarto)

# Create a vector of all states and the District of Columbia
state <- tibble(state.name) %>%
  rbind("District of Columbia") %>%
  pull(state.name)

# Create a tibble with information on the:
# input R Markdown document
# output HTML file
# parameters needed to knit the document
reports <- tibble(
❷ input = "urban-covid-budget-report.qmd",
  output_file = str_glue("{state}.xhtml"),
❸ execute_params = map(state, ~list(state = .))
)

# Generate all of our reports
reports %>%
❹ pwalk(quarto_render)
```
```

This updated *render.R* file loads the `quarto` package instead of the `rmarkdown` package ❶ and changes the input file to *urban-covid-budget-report.qmd* ❷. The `reports` tibble uses `execute_params` instead of `params` ❸ because this is the argument that the `quarto_render()` function expects. To render the reports, the `quarto_render()` function replaces the `render()` function from the `markdown` package ❹. As in Chapter 7, running this code should produce a report for each state.

## Making Presentations

Quarto can also produce slideshow presentations like those you made in Chapter 8 with the `xaringan` package. To make a presentation with Quarto, click **File ▸ New File ▸ Quarto Presentation**. Choose **Reveal JS** to make your slides and leave the Engine and Editor options untouched.

The slides you'll make use the `reveal.js` JavaScript library under the hood, a technique similar to making slides with `xaringan`. The following code updates the presentation you made in Chapter 8 so that it works with Quarto:

~~~
---
title: "Penguins Report"
author: "David Keyes"
format: revealjs
execute:
  echo: false
  warning: false
  message: false
---

# Introduction

```{r}
library(tidyverse)
```

```{r}
penguins <- read_csv("https://raw.githubusercontent.com/rfort
herestofus/r-without-statistics/
main/data/penguins-2008.csv")
```
~~~

~~~
We are writing a report about the **Palmer Penguins**. These
penguins are *really* amazing.
There are three species:

- Adelie
- Gentoo
- Chinstrap

## Bill Length

We can make a histogram to see the distribution of bill lengt
hs.

```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
~~~

```{r}
average_bill_length <- penguins %>%
  summarize(avg_bill_length = mean(
    bill_length_mm,
    na.rm = TRUE
  )) %>%
  pull(avg_bill_length)
```

The chart shows the distribution of bill lengths. The average
 bill length is
`r average_bill_length` millimeters.
```

This code sets `format: revealjs` in the YAML to make a presentation and adds several global code chunk options in the `execute` section. It then removes the three dashes used to make slide breaks because first- or second-level headings make new slides in Quarto (though you could still use three dashes to manually add slide breaks). When you render this code, you should get an HTML file with your slides. The output should look similar to the default `xaringan` slides from Chapter 8.
```

### *Revealing Content Incrementally*

Quarto slides can incrementally reveal content. To reveal bulleted and numbered lists one item at a time by default, add `incremental: true` to the document's YAML like so:

```
---
title: "Penguins Report"
author: "David Keyes"
format:
  revealjs:
    incremental: true
execute:
  echo: false
  warning: false
  message: false
---
```

As a result of this code, the content in all lists in the presentation should appear on the slide one item at a time.

You can also set just some lists to incrementally reveal using this format:

```
::: {.incremental}
- Adelie
- Gentoo
- Chinstrap
:::
```

Using `:::` to start and end a segment of the document creates a section in the resulting HTML file known as a *div*. The HTML `<div>` tag allows you to define properties within that section. In this code, adding `{.incremental}` sets a custom CSS class that displays the list incrementally.

### *Aligning Content and Adding Background Images*

You can use a `<div>` tag to create columns in Quarto slides, too. Say you want to create a slide with content in two columns, as in Figure 10-2.

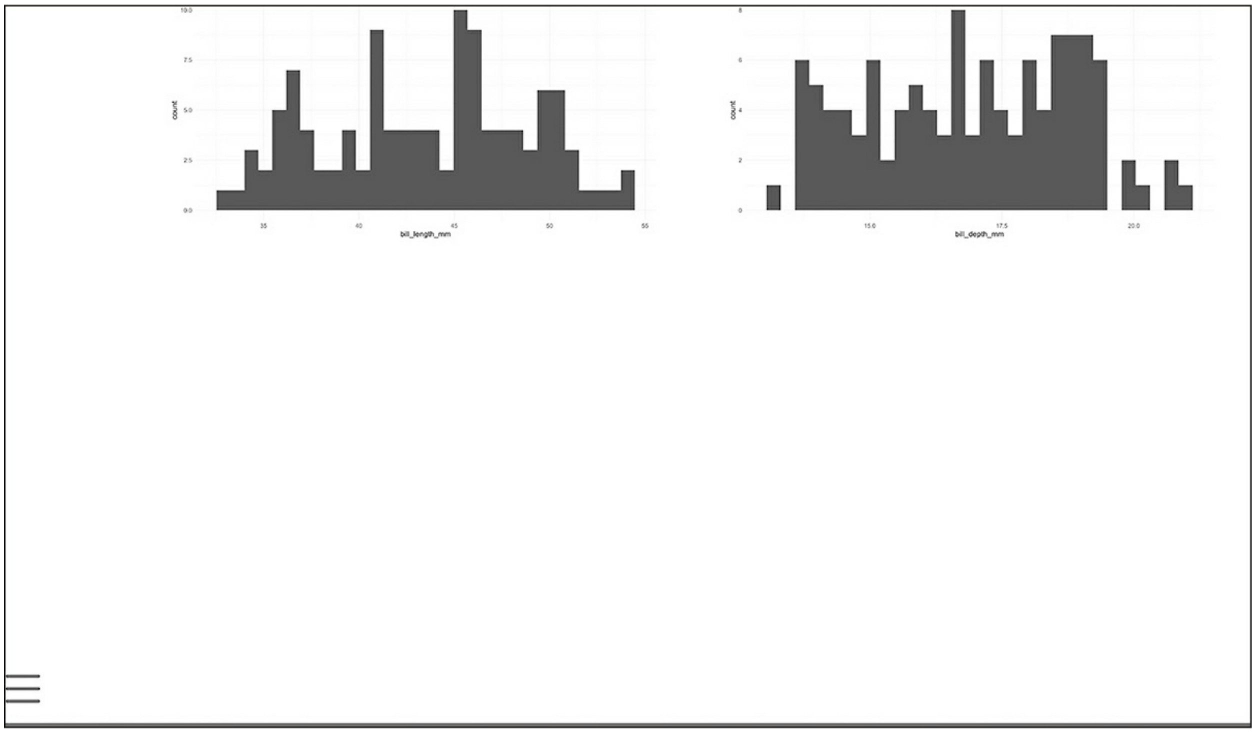


*Figure 10-2: Creating two columns with a <div> tag*

The following code creates this two-column slide:

~~~
:::: {.columns}

::: {.column width="50%"}
```{r}
penguins %>%
  ggplot(aes(x = bill_length_mm)) +
  geom_histogram() +
  theme_minimal()
~~~

:::

::: {.column width="50%"}
```{r}
penguins %>%
  ggplot(aes(x = bill_depth_mm)) +
  geom_histogram() +
  theme_minimal()
```
```
```

```
:::

::::
```

Notice the `:::`, as well as `::::`, which creates nested `<div>` sections. The `columns` class tells the HTML that all content within the `::::` should be laid out as columns. Then, `::: {.column width="50%"}` starts a `<div>` that takes up half the width of the slide. The closing `:::` and `::::` indicate the end of the section.

When using `xaringan`, you easily centered content on a slide by surrounding it with `.center[]`. Alignment in Quarto is slightly more complicated. Quarto has no built-in CSS class to center content, so you'll need to create one yourself. Begin a CSS code chunk and a custom class called `center-slide`:

~~~
```{css}
.center-slide {
    text-align: center;
}
```
~~~

This CSS center-aligns all content. (The `text-align` property aligns images, too, not just text.)

To apply the new `center-slide` class, put it next to the title of the slide, as follows:

```
## Bill Length {.center-slide}
```

With the custom CSS applied, the slide should now center all content.

Finally, when working in `xaringan`, you added a background image to a slide. To do the same thing in Quarto, apply the `background-image` attribute to a slide, like so:

```
## Penguins {background-image="penguins.jpg"}
```

This should add a slide with the text *Penguins* in front of the selected image.

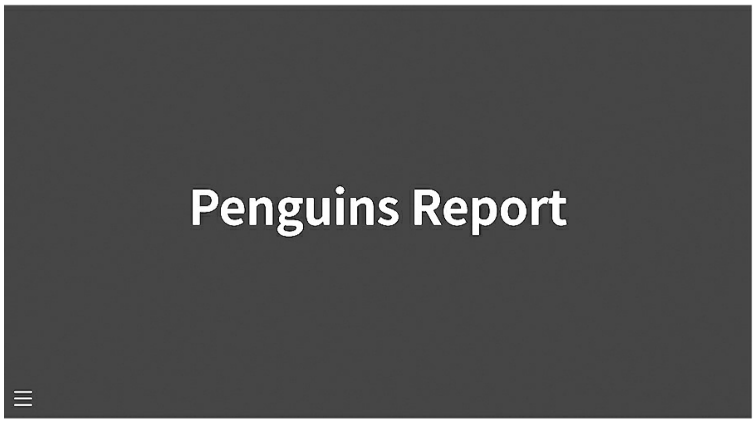
## *Customizing Your Slides with Themes and CSS*

You've started making some changes to the look and feel of the Quarto slides, but you can add even more customization to your design. As with `xaringan`, there are two main ways to further customize your slides in Quarto: using existing themes and changing the CSS.

Themes are the easiest way to change your slide design. To apply a theme in Quarto, simply add its name to your YAML:

```
---
title: "Penguins Report"
format:
  revealjs:
    theme: dark
---
```

Using this option should change the theme from light (the default) to dark. You can see the title slide with the dark theme applied in Figure 10-3. To see the full list of available themes, go to *https://quarto.org/docs/presentations/revealjs/themes.xhtml*.



Penguins Report

*Figure 10-3: A slide with the dark theme applied*

The second option to change your slide design further is to write custom CSS. Quarto uses a type of CSS called Sass that lets you include variables in the CSS. These variables resemble those from the `xaringanthemer` package, which allowed you to set values for header formatting using `header_h2_font_size` and `header_color`.

Go to **File ▸ New File ▸ New Text File**, create a Sass file called *theme.scss*, and add the following two mandatory sections:

```
/*-- scss:defaults --*/

/*-- scss:rules --*/
```

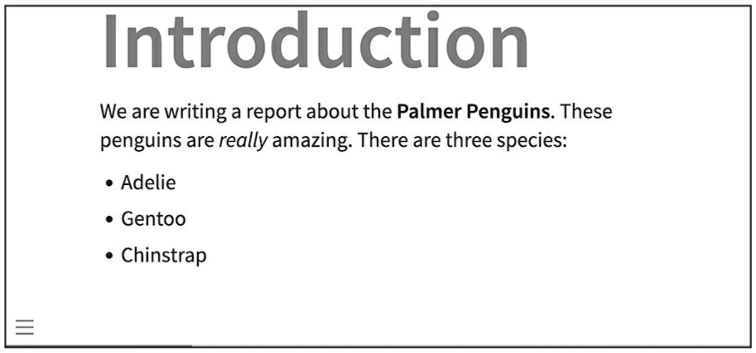
The `scss:defaults` section is where you use the Quarto Sass variables. For example, to change the color and size of first-level headers, add this code:

```
/*-- scss:defaults --*/
$presentation-heading-color: red;
$presentation-h1-font-size: 150px;

/*-- scss:rules --*/
```

All Quarto Sass variables start with a dollar sign, followed by a name. To apply these tweaks to your slides, adjust your YAML to tell Quarto to use the custom *theme.scss* file:

```
---
title: "Penguins Reports"
format:
  revealjs:
    theme: theme.scss
---
```

Figure 10-4 shows the changes applied to the rendered slides.



## Introduction

We are writing a report about the **Palmer Penguins.** These penguins are *really* amazing. There are three species:

- Adelie
- Gentoo
- Chinstrap

*Figure 10-4: A slide modified using custom CSS*

All predefined variables should go in the `scss:defaults` section. You can find the full list of these variables at *https://quarto.org/docs/presentations/revealjs/themes.xhtml#sass-variables*.

The `scss:rules` section is where you can add CSS tweaks for which there are no existing variables. For example, you could place the code you wrote to center the slide's content in this section:

```
/*-- scss:defaults --*/
$presentation-heading-color: red;
$presentation-h1-font-size: 150px;

/*-- scss:rules --*/
.center-slide {
  text-align: center;
}
```

Because rendered Quarto slides are HTML documents, you can tweak them however you'd like with custom CSS. What's more, because the slides use `reveal.js` under the hood, any features built into that JavaScript library work in Quarto. This library includes easy ways to add transitions, animations, interactive content, and much more. The demo Quarto presentation available at *https://quarto.org/docs/presentations/revealjs/demo/* shows many of these features in action.

## Making Websites

Quarto can make websites without requiring the use of an external package like `distill`. To create a Quarto website, go to **File ▸ New Project**. Select

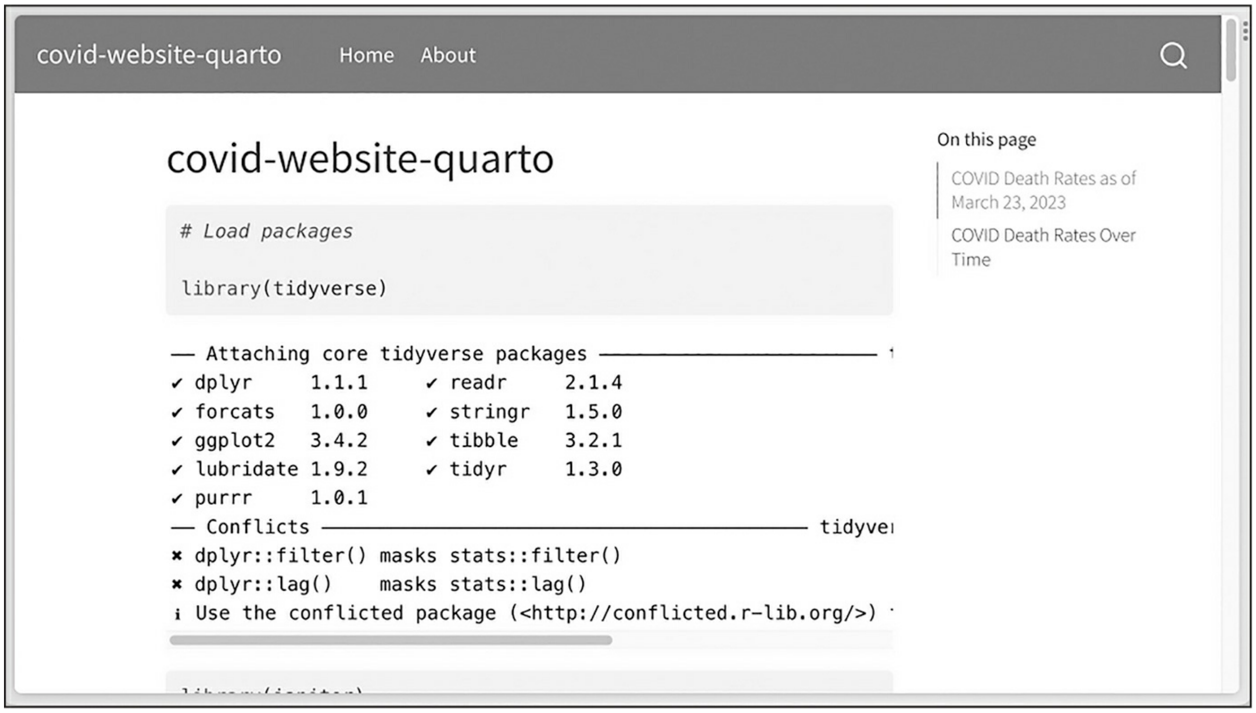
**New Directory**, then **Quarto Website**. You'll be prompted to choose a directory in which to place your project. Keep the default engine (Knitr), check **Create a Git Repository** (which should show up only if you've already installed Git), and leave everything else unchecked.

Click **Create Project**, which should create a series of files: *index.qmd*, *about.qmd*, *_quarto.yml*, and *styles.css*. These files resemble those created by the `distill` package. The *.qmd* files are where you'll add content, the *_quarto.yml* file is where you'll set options for the entire website, and the *styles.css* file is where you'll add CSS to customize the website's appearance.

## Building the Website

You'll start by modifying the *.qmd* files. Open the home page file (*index.qmd*), delete the default content after the YAML, and replace it with the content from the website you made in Chapter 9. Remove the `layout = "l-page"` element, which you used to widen the layout. I'll discuss how to change the page's layout in Quarto later in this section.

To render a Quarto website, look for the Build tab in the top right of RStudio and click **Render Website**. The rendered website should now appear in the Viewer pane on the bottom-right pane of RStudio. If you navigate to the Files pane on the same panel, you should also see that a *_site* folder has been created to hold the content of the rendered site. Try opening the *index.xhtml* file in your web browser. You should see the website in Figure 10-5.


covid-website-quarto
Home
About
On this page
COVID Death Rates as of March 23, 2023
COVID Death Rates Over Time

covid-website-quarto

```
# Load packages

library(tidyverse)
```

```
── Attaching core tidyverse packages ────────────────────────
✔ dplyr     1.1.1     ✔ readr     2.1.4
✔ forcats   1.0.0     ✔ stringr   1.5.0
✔ ggplot2   3.4.2     ✔ tibble    3.2.1
✔ lubridate 1.9.2     ✔ tidyr     1.3.0
✔ purrr     1.0.1
── Conflicts ──────────────────────────────────────── tidyve
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>)
```

*Figure 10-5: The Quarto website with warnings and messages*

As you can see, the web page includes many warnings and messages that you don’t want to show. In R Markdown, you removed these in the `setup` code chunk; in Quarto, you can do so in the YAML. Add the following code to the *index.qmd* YAML to remove all code, warnings, and messages from the output:

```
execute:
  echo: false
  warning: false
  message: false
```

Note, however, that these options will make changes to only one file. Next, you’ll see how to set these options for the entire website.

## Setting Options

When using `distill`, you modified the *_site.yml* file to make changes to all files in the website. In Quarto, you use the *_quarto.yml* file for the same purpose. If you open it, you should see three sections:

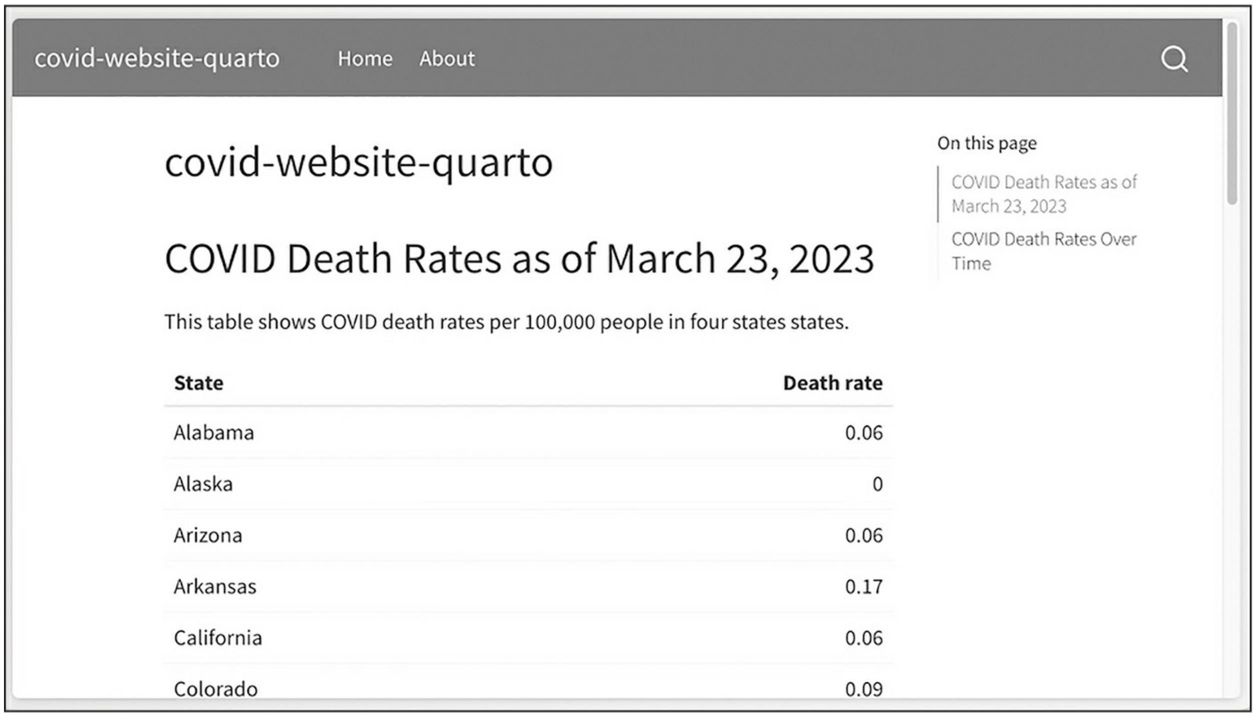
```
Project:
  type: website

website:
  title: "covid-website-quarto"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - about.qmd

format:
  html:
    theme: cosmo
    css: styles.css
    toc: true
```

The top section sets the project type (in this case, a website). The middle section defines the website's title and determines the options for its navigation bar. The bottom section modifies the site's appearance.

You'll start from the bottom. To remove code, warnings, and messages for every page in the website, add the portion of the YAML you wrote earlier to the *_quarto.yml* file. The bottom section should now look like this:

```
format:
  html:
    theme: cosmo
    css: styles.css
    toc: true
execute:
  echo: false
  warning: false
  message: false
```

If you build the website again, you should now see just the content, as in Figure 10-6.


covid-website-quarto
Home
About
On this page
COVID Death Rates as of March 23, 2023
COVID Death Rates Over Time

covid-website-quarto

COVID Death Rates as of March 23, 2023

This table shows COVID death rates per 100,000 people in four states states.

| State | Death rate |
|---|---|
| Alabama | 0.06 |
| Alaska | 0 |
| Arizona | 0.06 |
| Arkansas | 0.17 |
| California | 0.06 |
| Colorado | 0.09 |

*Figure 10-6: The website with warnings and messages removed*

In this section of the *_quarto.yml* file, you can add any options you would otherwise place in a single *.qmd* file to apply them across all the pages of your website.

## ***Changing the Website's Appearance***

The `format` section of the *_quarto.yml* file determines the appearance of rendered files. By default, Quarto applies a theme called `cosmo`, but there are many themes available. (You can see the full list at *https://quarto.org/docs/output-formats/html-themes.xhtml*.) To see how a different theme affects the output, make the following change:

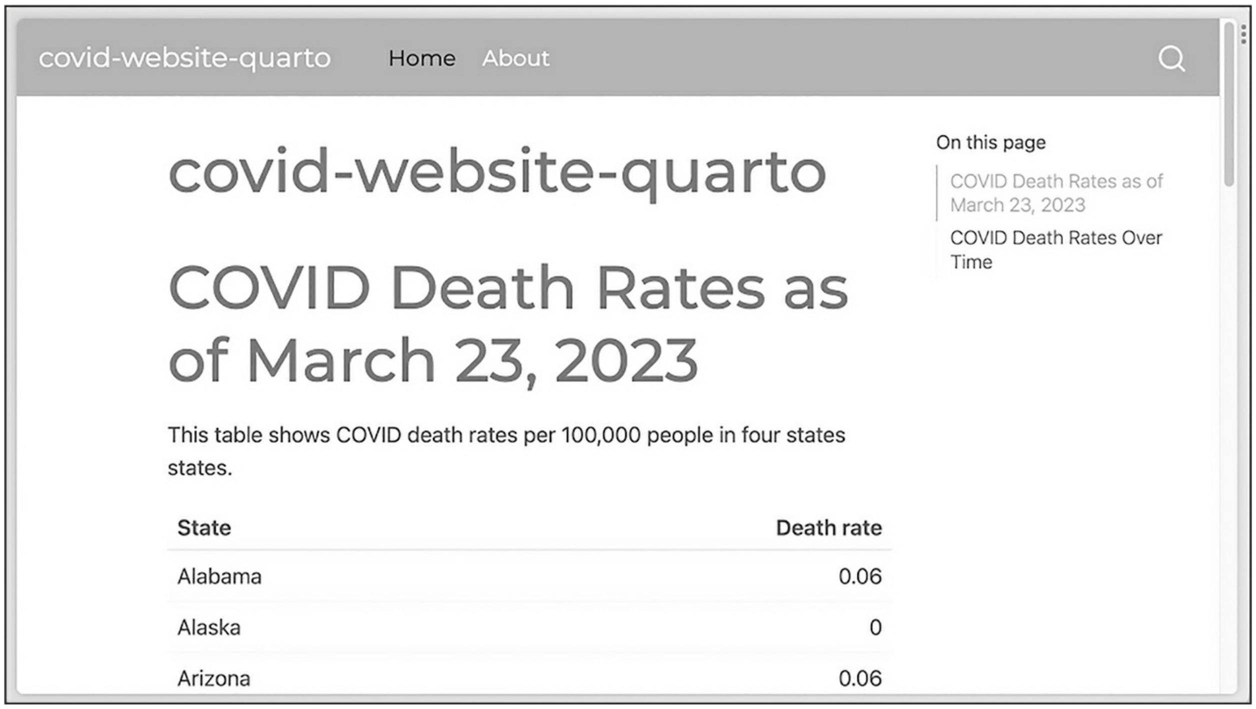
```
format:
  html:
    theme: minty
    css: styles.css
    toc: true
```

The `minty` theme changes the website's fonts and updates the color scheme to gray and light green.

In addition to using prebuilt themes, you can customize your website with CSS. The `css: styles.css` section in the *_quarto.yml* file indicates that Quarto will use any CSS in the *styles.css* file when rendering. Try adding the following CSS to *styles.css* to make first-level headers red and 50 pixels large:

```
h1 {
  color: red;
  font-size: 50px;
}
```

The re-rendered *index.xhtml* now has large red headings (shown in grayscale in Figure 10-7).


covid-website-quarto
Home
About
On this page
COVID Death Rates as of March 23, 2023
COVID Death Rates Over Time

covid-website-quarto

COVID Death Rates as of March 23, 2023

This table shows COVID death rates per 100,000 people in four states states.

| State | Death rate |
|---|---|
| Alabama | 0.06 |
| Alaska | 0 |
| Arizona | 0.06 |

*Figure 10-7: The website with custom CSS applied*

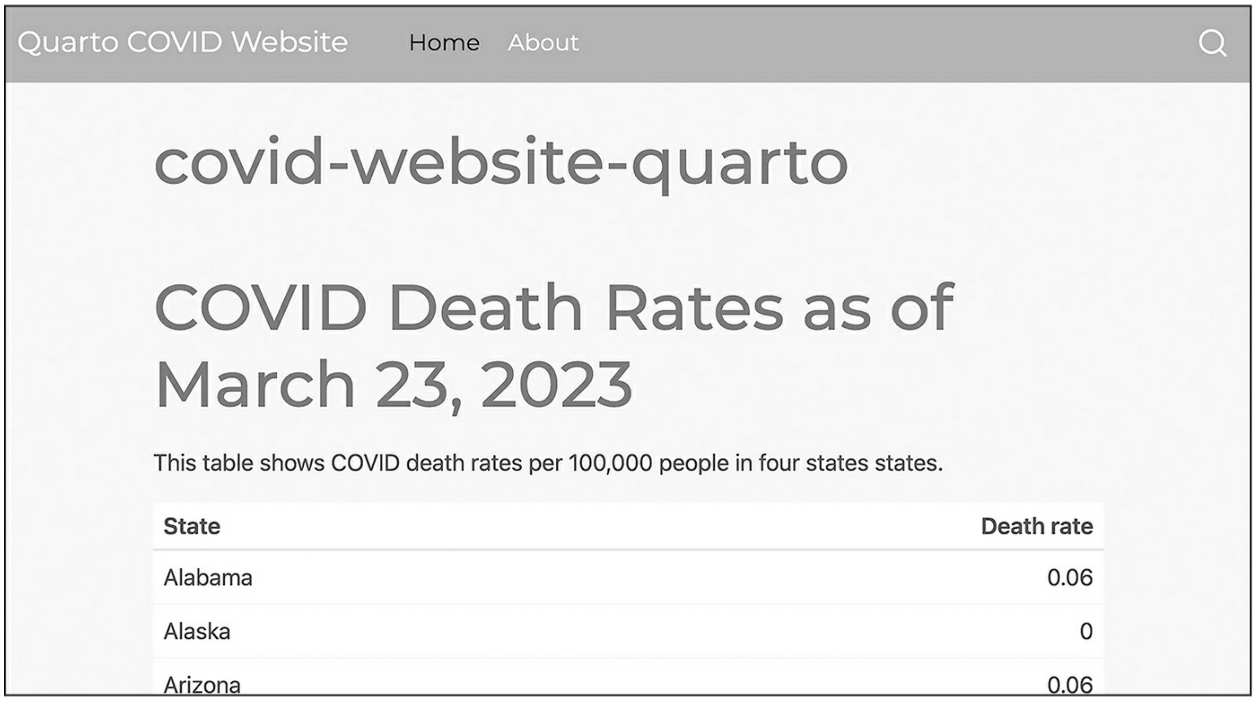
An alternative approach to customizing your website is to use Sass variables in a *.scss* file, as you did in your presentation. For example, create a file called *styles.scss* and add a line like this one to make the body background bright yellow:

```
/*-- scss:defaults --*/
$body-bg: #eeeeee;
```

To get Quarto to use the *styles.scss* file, adjust the `theme` line as follows:

```
format:
  html:
    theme: [minty, styles.scss]
    css: styles.css
    toc: true
```

This syntax tells Quarto to use the `minty` theme, then make additional tweaks based on the *styles.scss* file. If you render the website again, you should see the bright yellow background throughout (Figure 10-8, again in grayscale for print).



covid-website-quarto

COVID Death Rates as of March 23, 2023

This table shows COVID death rates per 100,000 people in four states states.

| State | Death rate |
|---|---|
| Alabama | 0.06 |
| Alaska | 0 |
| Arizona | 0.06 |

*Figure 10-8: The website with custom CSS applied through* styles.scss

Note that when you add a *.scss* file, the tweaks made in *styles.css* no longer apply. If you wanted to use those, you'd need to add them to the *styles.scss* file.

The line `toc: true` creates a table of contents on the right side of the web pages (which you can see in Figures 10-5 through 10-7, labeled On This Page). You can remove the table of contents by changing `true` to `false`. Add any further options, such as figure height, to the bottom section of the *_quarto.yml* file.

## Adjusting the Title and Navigation Bar

The middle section of the *_quarto.yml* file sets the website's title and navigation. Change the title and the text for the About page link as follows:

```
website:
  title: "Quarto COVID Website"
  navbar:
    left:
      - href: index.qmd
        text: Home
      - href: about.qmd
        text: About This Website
```

Changing the title requires adjusting the `title` line. The `navbar` section functions nearly identically to how it does with `distill`. The `href` line lists the files the navigation bar should link to. The optional `text` line specifies the text that should show up for that link. Figure 10-9 shows these changes applied to the website.

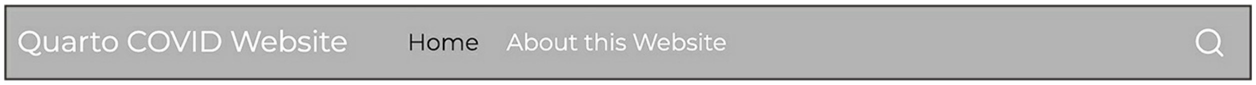


*Figure 10-9: The changes to the navigation bar*

The title on the home page is still covid-website-quarto, but you could change this in the *index.qmd* file.

## Creating Wider Layouts

When you created a website with `distill`, you used the line `layout = "l-page"` to widen the map on the web page. You can accomplish the same result with Quarto by using the `:::` syntax to add HTML `<div>` tags:

~~~
:::{.column-screen-inset}
```{r}
#| out-width: 100%
# Make map

most_recent <- us_states %>%
  left_join(covid_data, by = "state") %>%
  slice_max(
    order_by = date,
    n = 1
  )

most_recent %>%
  ggplot(aes(fill = deaths_avg_per_100k)) +
  geom_sf() +
  scale_fill_viridis_c(option = "rocket") +
  labs(fill = "Deaths per\n100,000 people") +
  theme_void()
```
:::
~~~

This code adds `:::{.column-screen-inset}` to the beginning of the mapmaking code chunk and `:::` to the end of it. This code chunk now also includes the line `#| out-width: 100%` to specify that the map should take up all of the available width. Without this line, the map would take up only a portion of the window. There are a number of different output widths you can use; see the full list at *https://quarto.org/docs/authoring/article-layout.xhtml*.

### *Hosting Your Website on GitHub Pages and Quarto Pub*

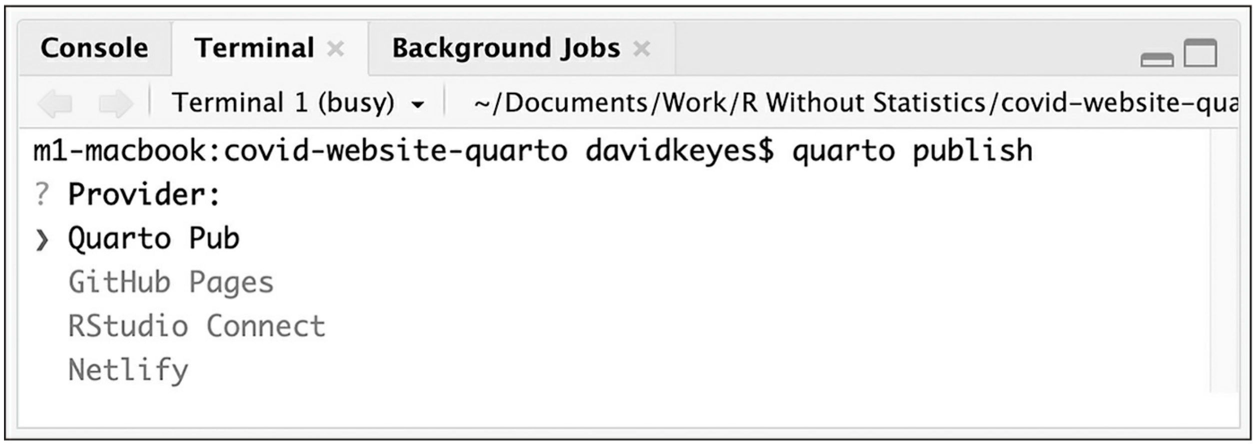
You can host your Quarto website using GitHub Pages, just as you did with your `distill` website. Recall that GitHub Pages requires you to save the website's files in the *docs* folder. Change the *_quarto.yml* file so that the site outputs to this folder:

```
project:
  type: website
  output-dir: docs
```

Now, when you render the site, the HTML and other files should show up in the *docs* directory. At this point, you can push your repository to

GitHub, adjust the GitHub Pages settings as you did in Chapter 9, and see the URL at which your Quarto website will live.

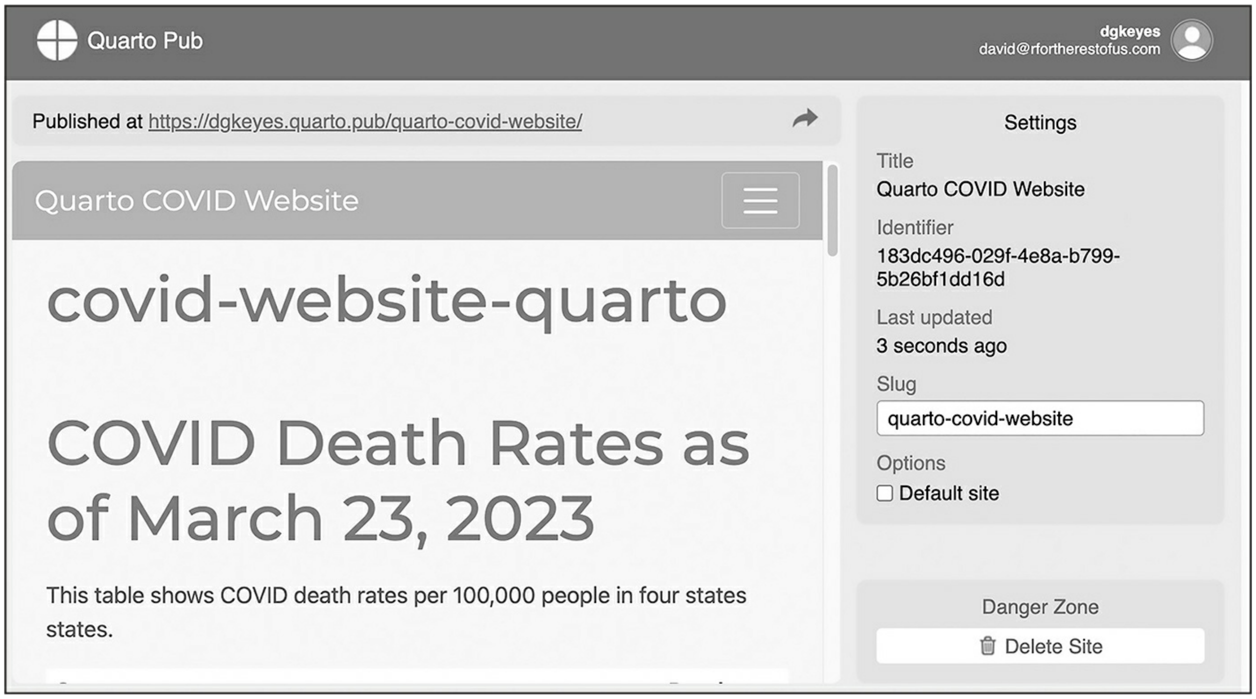
As an alternative to GitHub Pages, Quarto has a free service called Quarto Pub that makes it easy to get your materials online. If you're not a GitHub user, this is a great way to publish your work. To see how it works, you'll publish the website you just made to it. Click the **Terminal** tab on the bottom-left pane of RStudio. At the prompt, enter **`quarto publish`**. This should bring up a list of ways you can publish your website, as shown in Figure 10-10.



```
m1-macbook:covid-website-quarto davidkeyes$ quarto publish
? Provider:
❯ Quarto Pub
  GitHub Pages
  RStudio Connect
  Netlify
```

*Figure 10-10: The list of providers to publish your Quarto website*

Press **Enter** to select Quarto Pub. You'll then be asked to authorize RStudio to publish to Quarto Pub. Enter **Y** to do so, which should take you to *https://quartopub.com*. Sign up for an account (or sign in if you already have one). You should see a screen indicating that you have successfully signed in and authorized RStudio to connect with Quarto Pub. From there, you can return to RStudio, which should prompt you to select a name for your website. The easiest option is to use your project's name. Once you enter the name, Quarto Pub should publish the site and take you to it, as shown in Figure 10-11.


Quarto Pub

dgkeyes
david@rfortherestofus.com

Published at https://dgkeyes.quarto.pub/quarto-covid-website/

Quarto COVID Website

covid-website-quarto

COVID Death Rates as of March 23, 2023

This table shows COVID death rates per 100,000 people in four states states.

Settings

Title
Quarto COVID Website

Identifier
183dc496-029f-4e8a-b799-5b26bf1dd16d

Last updated
3 seconds ago

Slug
quarto-covid-website

Options
☐ Default site

Danger Zone
Delete Site

*Figure 10-11: The website published on Quarto Pub*

When you make updates to your site, you can republish it to Quarto Pub using the same steps. Quarto Pub is probably the easiest way to publish HTML files made with Quarto.

## Summary

As you've seen in this chapter, you can do everything you did in R Markdown using Quarto, without loading any external packages. In addition, Quarto's different output formats use a more consistent syntax. For example, because you can make new slides in Quarto by adding first- or second-level headers, the Quarto documents you use to create reports should translate easily to presentations.

You're probably wondering at this point whether you should use R Markdown or Quarto. It's a good question, and one many in the R community are thinking about. R Markdown isn't going away, so if you already use it, you don't need to switch. If you're new to R, however, you may be a good candidate for Quarto, as its future features may not be backported to R Markdown.

Ultimately, the differences between R Markdown and Quarto are relatively small, and the impact of switching between tools should be minor. Both R Markdown and Quarto can help you become more efficient, avoid manual errors, and share results in a wide variety of formats.

## Additional Resources

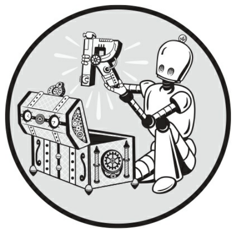
- Andrew Bray, Rebecca Barter, Silvia Canelón, Christophe Dervieu, Devin Pastor, and Tatsu Shigeta, “From R Markdown to Quarto,” workshop materials from rstudio::conf 2022, Washington, DC, July 25–26, 2022, *https://rstudio-conf-2022.github.io/rmd-to-quarto/*.
- Tom Mock, “Getting Started with Quarto,” online course, accessed December 1, 2023, *https://jthomasmock.github.io/quarto-in-two-hours/*.

# PART III

# AUTOMATION AND COLLABORATION

# 11

# AUTOMATICALLY ACCESSING ONLINE DATA



So far, you've imported data into your projects from CSV files. Many online datasets allow you to export CSVs, but before you do so, you should look for packages to automate your data access. If you can eliminate the manual steps involved in fetching data, your analysis and reporting will be more accurate. You'll also be able to efficiently update your report when the data changes.

R offers many ways to automate the process of accessing online data. In this chapter, I'll discuss two such approaches. First, you will use the `googlesheets4` package to fetch data directly from Google Sheets. You'll learn how to connect your R Markdown project to Google so you can automatically download data when a Google Sheet updates. Then, you'll use the `tidycensus` package to access data from the US Census Bureau. You'll work with two large census datasets, the Decennial Census and the American Community Survey, and practice visualizing them.

## Importing Data from Google Sheets with googlesheets4

By using the `googlesheets4` package to access data directly from Google Sheets, you avoid having to manually download data, copy it into your project, and adjust your code so it imports that new data every time you want to update a report. This package lets you write code that automatically fetches new data directly from Google Sheets. Whenever you need to update your report, you can simply run your code to refresh the data. In addition, if you work with Google Forms, you can pipe your data into Google Sheets, completely automating the workflow from data collection to data import.

Using the `googlesheets4` package can help you manage complex datasets that update frequently. For example, in her role at the Primary Care Research Institute at the University of Buffalo, Meghan Harris used it for a research project about people affected by opioid use disorder. The data came from a variety of surveys, all of which fed into a jumble of Google Sheets. Using `googlesheets4`, Harris was able to collect all of her data in one place and use R to put it to use. Data that had once been largely unused because accessing it was so complicated could now inform research on opioid use disorder.

This section demonstrates how the `googlesheets4` package works using a fake dataset about video game preferences that Harris created to replace her opioid survey data (which, for obvious reasons, is confidential).

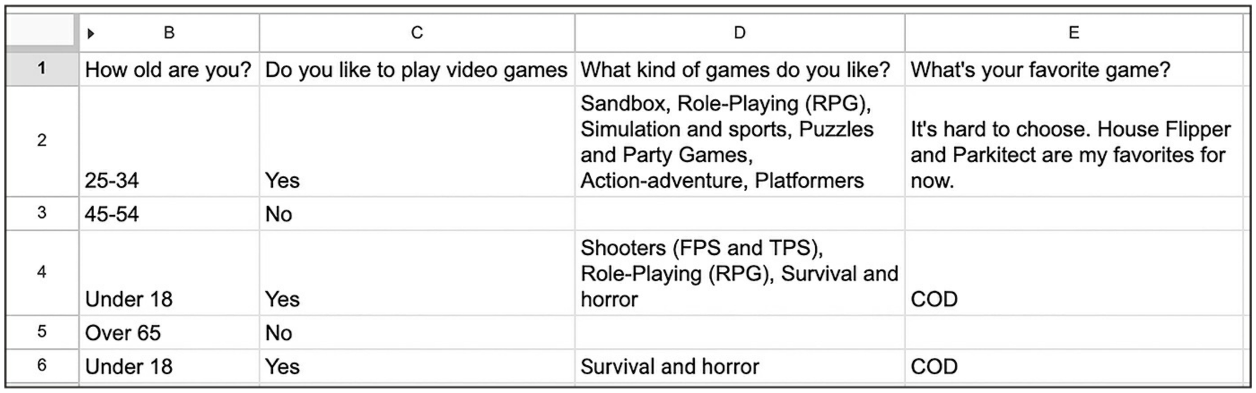
### *Connecting to Google*

To begin, install the `googlesheets4` package by running **`install.packages("googlesheets4")`**. Next, connect to your Google account by running the **`gs4_auth()`** function in the console. If you have more than one Google account, select the account that has access to the Google Sheet you want to work with.

Once you do so, a screen should appear. Check the box next to **See, Edit, Create, and Delete All Your Google Sheets Spreadsheets**. This will ensure that R can access data from your Google Sheets account. Click **Continue**, and you should see the message “Authentication complete. Please close this page and return to R.” The `googlesheets4` package will now save your credentials so that you can use them in the future without having to reauthenticate.

### *Reading Data from a Sheet*

Now that you’ve connected R to your Google account, you can import the fake data that Harris created about video game preferences (access it at *https://data.rfortherestofus.com/google-sheet*). Figure 11-1 shows what it looks like in Google Sheets.



| | B | C | D | E |
|---|---|---|---|---|
| 1 | How old are you? | Do you like to play video games | What kind of games do you like? | What's your favorite game? |
| 2 | 25-34 | Yes | Sandbox, Role-Playing (RPG), Simulation and sports, Puzzles and Party Games, Action-adventure, Platformers | It's hard to choose. House Flipper and Parkitect are my favorites for now. |
| 3 | 45-54 | No | | |
| 4 | Under 18 | Yes | Shooters (FPS and TPS), Role-Playing (RPG), Survival and horror | COD |
| 5 | Over 65 | No | | |
| 6 | Under 18 | Yes | Survival and horror | COD |

*Figure 11-1: The video game data in Google Sheets*

The `googlesheets4` package has a function called `read_sheet()` that allows you to pull in data directly from a Google Sheet. Import the data by passing the spreadsheet's URL to the function like so:

```
library(googlesheets4)

survey_data_raw <- read_sheet("https://docs.google.com/spread
sheets/d/
1AR0_RcFBg8wdiY4Cj-k8vRypp_txh27MyZuiRdqScog/edit?usp=sharing
")
```

Take a look at the `survey_data_raw` object to confirm that the data was imported. Using the `glimpse()` function from the `dplyr` package makes it easier to read:

```
library(tidyverse)

survey_data_raw %>%
  glimpse()
```

The `glimpse()` function, which creates one output row per variable, shows that you've successfully imported the data directly from Google

Sheets:

```
#> Rows: 5
#> Columns: 5
#> $ Timestamp                            <dttm> 05-16 15:20:50
#> $ `How old are you?`                   <chr> "25-34", "45-54
"...
#> $ `Do you like to play video games?` <chr> "Yes", "No", "Y
e...
#> $ `What kind of games do you like?`  <chr> "Sandbox, Role-
P...
#> $ `What's your favorite game?`       <chr> "It's hard to c
h...
```

Once you have the data in R, you can use the same workflow you've been using to create reports with R Markdown.

### *Using the Data in R Markdown*

The following code is taken from an R Markdown report that Harris made to summarize the video games data. You can see the YAML, the `setup` code chunk, a code chunk that loads packages, and the code to import data from Google Sheets:

~~~
---
title: "Video Game Survey"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = FALSE,
                      warning = FALSE,
                      message = FALSE)
```


```{r}
library(tidyverse)
library(janitor)
library(googlesheets4)
library(gt)
```
~~~

~~~
```{r}
# Import data from Google Sheets
❶ survey_data_raw <- read_sheet("https://docs.google.com/spread
sheets/d/
1AR0_RcFBg8wdiY4Cj-k8vRypp_txh27MyZuiRdqScog/edit?usp=sharing
")
```
~~~

This R Markdown document resembles those discussed in previous chapters, except for the way you import the data ❶. Because you're bringing it in directly from Google Sheets, there's no risk of, say, accidentally reading in the wrong CSV. Automating this step reduces the risk of error.

The next code chunk cleans the `survey_data_raw` object, saving the result as `survey_data_clean`:

~~~
```{r}
# Clean data
survey_data_clean <- survey_data_raw %>%
  clean_names() %>%
  mutate(participant_id = as.character(row_number())) %>%
  rename(
    age = how_old_are_you,
    like_games = do_you_like_to_play_video_games,
    game_types = what_kind_of_games_do_you_like,
    favorite_game = whats_your_favorite_game
  ) %>%
  relocate(participant_id, .before = age) %>%
  mutate(age = factor(age, levels = c("Under 18", "18-24", "2
5-34",
"35-44", "45-54", "55-64", "Over 65")))
```
~~~

Here, the `clean_names()` function from the `janitor` package makes the variable names easier to work with. Defining a `participant_id` variable using the `row_number()` function then adds a consecutively increasing number to each row, and the `as.character()` function makes the number a character. Next, the code changes several variable names with the `rename()` function. The `mutate()` function then transforms the `age` variable into a data structure known as a *factor*, which ensures that `age` will show up in the right order in your chart. Finally, the `relocate()` function positions `participant_id` before the `age` variable.

Now you can use the `glimpse()` function again to view your updated `survey_data_clean` data frame, which looks like this:

```
#> Rows: 5
#> Columns: 6
#> $ timestamp      <dttm> 2024-05-16 15:20:50, 2024-05-16 15
:21:28, 2024-05...
#> $ participant_id <chr> "1", "2", "3", "4", "5"
#> $ age            <fct> 25-34, 45-54, Under 18, Over 65, Un
...
#> $ like_games     <chr> "Yes", "No", "Yes", "No", "Yes"
#> $ game_types     <chr> "Sandbox, Role-Playing (RPG), Simul
...
#> $ favorite_game  <chr> "It's hard to choose. House Flipper
...
```

The rest of the report uses this data to highlight various statistics:

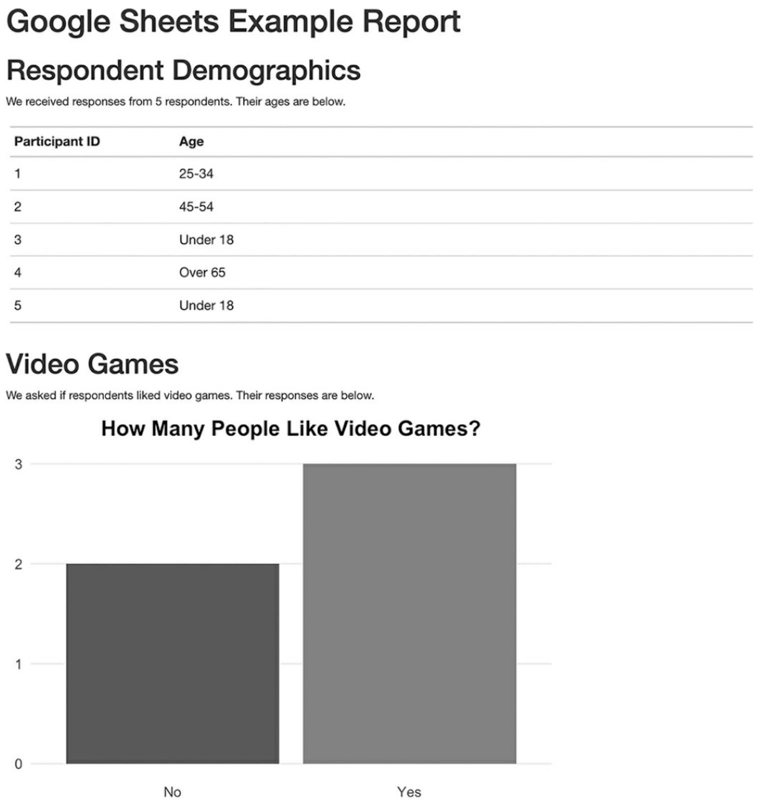
~~~
# Respondent Demographics

```{r}
# Calculate number of respondents
number_of_respondents <- nrow(survey_data_clean)❶
~~~

We received responses from `r number_of_respondents` responde
nts. Their ages are below.

```{r}
survey_data_clean %>%
  select(participant_id, age) %>%
  gt() %>%❷
  cols_label(
    participant_id = "Participant ID",
    age = "Age"
  ) %>%
  tab_style(
    style = cell_text(weight = "bold"),
    locations = cells_column_labels()
  ) %>%
```

~~~
  cols_align(
    align = "left",
    columns = everything()
  ) %>%
  cols_width(
    participant_id ~ px(200),
    age ~ px(700)
  )
```

# Video Games

We asked if respondents liked video games. Their responses ar
e below.

```{r}
survey_data_clean %>%
  count(like_games) %>%
  ggplot(aes(
    x = like_games, ❸
    y = n,
    fill = like_games
  )) +
  geom_col() +
  scale_fill_manual(values = c(
    "No" = "#6cabdd",
    "Yes" = "#ff7400"
  )) +
  labs(
    title = "How Many People Like Video Games?",
    x = NULL,
    y = "Number of Participants"
  ) +
  theme_minimal(base_size = 16) +
  theme(
    legend.position = "none",
    panel.grid.minor = element_blank(),
    panel.grid.major.x = element_blank(),
    axis.title.y = element_blank(),
    plot.title = element_text(
      face = "bold",
      hjust = 0.5
    )
  )
```
~~~

These sections calculate the number of survey respondents ❶, then put this value in the text using inline R code; create a table that breaks down the respondents by age group ❷; and generate a graph displaying how many respondents like video games ❸. Figure 11-2 shows the resulting report.



Google Sheets Example Report

Respondent Demographics

We received responses from 5 respondents. Their ages are below.

| Participant ID | Age |
|---|---|
| 1 | 25-34 |
| 2 | 45-54 |
| 3 | Under 18 |
| 4 | Over 65 |
| 5 | Under 18 |

Video Games

We asked if respondents liked video games. Their responses are below.

*Figure 11-2: The rendered video game report*

You can rerun the code at any point to fetch updated data. The survey had five responses today, but if you run it again tomorrow and it has additional responses, they will be included in the import. If you used Google Forms to run your survey and saved the results to a Google Sheet, you could produce this up-to-date report simply by clicking the Knit button in RStudio.

## *Importing Only Certain Columns*

In the previous sections, you read the data of the entire Google Sheet, but you also have the option to import only a section of a sheet. For example, the survey data includes a `timestamp` column. This variable is added automatically whenever someone submits a Google Form that pipes data into a Google Sheet, but you don't use it in your analysis, so you could get rid of it.

To do so, use the `range` argument in the `read_sheet()` function when importing the data like so:

```
read_sheet("https://docs.google.com/spreadsheets/d/1AR0_RcFBg
8wdiY4Cj-k8vRypp_
txh27MyZuiRdqScog/edit?usp=sharing",
          range = "Sheet1!B:E") %>%
  glimpse()
```

This argument lets you specify a range of data to import. It uses the same syntax you may have used to select columns in Google Sheets. In this example, `range = "Sheet1!B:E"` imports columns B through E (but not A, which contains the timestamp). Adding `glimpse()` and then running this code produces output without the `timestamp` variable:

```
#> Rows: 5
#> Columns: 4
#> $ `How old are you?`                <chr> "25-34", "45-54
"...
#> $ `Do you like to play video games?` <chr> "Yes", "No", "Y
e...
#> $ `What kind of games do you like?`  <chr> "Sandbox, Role-
P...
#> $ `What's your favorite game?`       <chr> "It's hard to c
h...
```

There are a number of other useful functions in the `googlesheets4` package. For example, if you ever need to write your output back to a Google Sheet, the `write_sheet()` function is there to help. To explore other functions in the package, check out its documentation website at *https://googlesheets4.tidyverse.org/index.xhtml*.

Now we'll turn our attention to another R package that allows you to automatically fetch data, this time from the US Census Bureau.

## Accessing Census Data with tidycensus

If you’ve ever worked with data from the US Census Bureau, you know what a hassle it can be. Usually, the process involves visiting the Census Bureau website, searching for the data you need, downloading it, and then analyzing it in your tool of choice. This pointing and clicking gets very tedious after a while.

Kyle Walker, a geographer at Texas Christian University, and Matt Herman (creator of the Westchester COVID-19 website discussed in Chapter 9) developed the `tidycensus` package to automate the process of importing Census Bureau data into R. With `tidycensus`, you can write just a few lines of code to get data about, say, the median income in all counties in the United States.

In this section, I’ll show you how the `tidycensus` package works using examples from two datasets to which it provides access: the Decennial Census, administered every 10 years, and the annual American Community Survey. I’ll also show you how to use the data from these two sources to perform additional analysis and make maps by accessing geospatial and demographic data simultaneously.

## *Connecting to the Census Bureau with an API Key*

Begin by installing `tidycensus` using **`install.packages("tidycensus")`**. To use `tidycensus`, you must get an application programming interface (API) key from the Census Bureau. *API keys* are like passwords that online services use to determine whether you’re authorized to access data.

To obtain this key, which is free, go to *https://api.census.gov/data/key_signup.xhtml* and enter your details. Once you receive the key by email, you need to put it in a place where `tidycensus` can find it. The `census_api_key()` function does this for you, so after loading the `tidycensus` package, run the function as follows, replacing *`123456789`* with your actual API key:

```
library(tidycensus)

census_api_key("123456789", install = TRUE)
```

The `install = TRUE` argument saves your API key in your *.Renviron* file, which is designed for storing confidential information. The package will look for your API key there in the future so that you don't have to reenter it every time you use the package.

Now you can use `tidycensus` to access Census Bureau datasets. While the Decennial Census and the American Community Survey are the most common, Chapter 2 of Kyle Walker's book *Analyzing US Census Data: Methods, Maps, and Models in R* discusses others you can access.

## *Working with Decennial Census Data*

The `tidycensus` packages includes several functions dedicated to specific Census Bureau datasets, such as `get_decennial()` for Decennial Census data. To access data from the 2020 Decennial Census about the Asian population in each state, use the `get_decennial()` function with three arguments as follows:

```
get_decennial(geography = "state",
              variables = "P1_006N",
              year = 2020)
```

Setting the `geography` argument to `"state"` tells `get_decennial()` to access data at the state level. In addition to the 50 states, it will return data for the District of Columbia and Puerto Rico. The `variables` argument specifies the variable or variables you want to access. Here, `P1_006N` is the variable name for the total Asian population. I'll discuss how to identify other variables you may want to use in the next section. Finally, `year` specifies the year for which you want to access data—in this case, 2020.

Running this code returns the following:

```
#> # A tibble: 52 × 4
#>    GEOID NAME                 variable   value
#>    <chr> <chr>                <chr>      <dbl>
#>  1 42    Pennsylvania         P1_006N   510501
#>  2 06    California           P1_006N  6085947
#>  3 54    West Virginia        P1_006N    15109
#>  4 49    Utah                 P1_006N    80438
#>  5 36    New York             P1_006N  1933127
#>  6 11    District of Columbia P1_006N    33545
#>  7 02    Alaska               P1_006N    44032
#>  8 12    Florida              P1_006N   643682
#>  9 45    South Carolina       P1_006N    90466
#> 10 38    North Dakota         P1_006N    13213
--snip--
```

The resulting data frame has four variables. `GEOID` is the geographic identifier assigned to the state by the Census Bureau. Each state has a geographic identifier, as do all counties, census tracts, and other geographies. `NAME` is the name of each state, and `variable` is the name of the variable you passed to the `get_decennial()` function. Finally, `value` is the numeric value for the state and variable in each row. In this case, it represents the total Asian population in each state.

## *Identifying Census Variable Values*

You've just seen how to retrieve the total number of Asian residents of each state, but say you want to calculate that number instead as a percentage of all the state's residents. To do that, first you need to retrieve the variable for the state's total population.

The `tidycensus` package has a function called `load_variables()` that shows all of the variables from a Decennial Census. Run it with the `year` argument set to `2020` and `dataset` set to `pl` as follows:

```
load_variables(year = 2020,
               dataset = "pl")
```

Running this code pulls data from so-called redistricting summary data files (which Public Law 94-171 requires the Census Bureau to produce every 10 years) and returns the name, label (description), and concept (category) of all available variables:

```
#> # A tibble: 301 × 3
#>    name    label                                  con
cept
#>    <chr>   <chr>                                  <ch
r>
#>  1 H1_001N " !!Total:"                            OCC
U...
#>  2 H1_002N " !!Total:!!Occupied"                  OCC
U...
```

```
#>  3 H1_003N " !!Total:!!Vacant"                         OCC
U...
#>  4 P1_001N " !!Total:"                                 RAC
E
#>  5 P1_002N " !!Total:!!Population of one race:"        RAC
E
#>  6 P1_003N " !!Total:!!Population of one race:!!Whi... RAC
E
#>  7 P1_004N " !!Total:!!Population of one race:!!Bla... RAC
E
#>  8 P1_005N " !!Total:!!Population of one race:!!Ame... RAC
E
#>  9 P1_006N " !!Total:!!Population of one race:!!Asi... RAC
E
#> 10 P1_007N " !!Total:!!Population of one race:!!Nat... RAC
E
--snip--
```

By looking at this list, you can see that the variable `P1_001N` returns the total population.

## *Using Multiple Census Variables*

Now that you know which variables you need, you can use the `get_decennial()` function again with two variables at once:

```
get_decennial(geography = "state",
              variables = c("P1_001N", "P1_006N"),
              year = 2020) %>%
  arrange(NAME)
```

Adding `arrange(NAME)` after `get_decennial()` sorts the results by state name, allowing you to easily see that the output includes both variables for each state:

```
#> # A tibble: 104 × 4
#>    GEOID NAME       variable    value
#>    <chr> <chr>      <chr>       <dbl>
#>  1 01    Alabama    P1_001N   5024279
#>  2 01    Alabama    P1_006N     76660
#>  3 02    Alaska     P1_001N    733391
#>  4 02    Alaska     P1_006N     44032
#>  5 04    Arizona    P1_001N   7151502
#>  6 04    Arizona    P1_006N    257430
#>  7 05    Arkansas   P1_001N   3011524
#>  8 05    Arkansas   P1_006N     51839
#>  9 06    California P1_001N  39538223
#> 10 06    California P1_006N   6085947
--snip--
```

When you're working with multiple census variables like this, you might have trouble remembering what names like `P1_001N` and `P1_006N` mean. Fortunately, you can adjust the code in the call to `get_decennial()` to give these variables more meaningful names using the following syntax:

```
get_decennial(geography = "state",
              variables = c(total_population = "P1_001N",
                            asian_population = "P1_006N"),
              year = 2020) %>%
  arrange(NAME)
```

Within the `variables` argument, this code specifies the new names for the variables, followed by the equal sign and the original variable names. The `c()` function allows you to rename multiple variables at one time.

Now it's much easier to see which variables you're working with:

```
#> # A tibble: 104 × 4
#>    GEOID NAME       variable            value
#>    <chr> <chr>      <chr>               <dbl>
#>  1 01    Alabama    total_population  5024279
#>  2 01    Alabama    asian_population    76660
#>  3 02    Alaska     total_population   733391
#>  4 02    Alaska     asian_population    44032
#>  5 04    Arizona    total_population  7151502
#>  6 04    Arizona    asian_population   257430
#>  7 05    Arkansas   total_population  3011524
#>  8 05    Arkansas   asian_population    51839
#>  9 06    California total_population 39538223
#> 10 06    California asian_population  6085947
#> # ... with 94 more rows
```

Instead of `P1_001N` and `P1_006N`, the variables appear as `total_population` and `asian_population`. Much better!

## *Analyzing Census Data*

Now you have the data you need to calculate the Asian population in each state as a percentage of the total. There are just a few functions to add to the code from the previous section:

```
get_decennial(
  geography = "state",
  variables = c(
    total_population = "P1_001N",
    asian_population = "P1_006N"
  ),
  year = 2020
) %>%
  arrange(NAME) %>%
  group_by(NAME) %>%
  mutate(pct = value / sum(value)) %>%
  ungroup() %>%
  filter(variable == "asian_population")
```

The `group_by(NAME)` function creates one group for each state because you want to calculate the Asian population percentage in each state (not for the entire United States). Then `mutate()` calculates each percentage, taking the `value` in each row and dividing it by the `total_population` and `asian_population` rows for each state. The `ungroup()` function removes the state-level grouping, and `filter()` shows only the Asian population percentage.

When you run this code, you should see both the total Asian population and the Asian population as a percentage of the total population in each state:

```
#> # A tibble: 52 × 5
#>    GEOID NAME                 variable          value
 pct
#>    <chr> <chr>                <chr>             <dbl>    <
dbl>
#>  1 01    Alabama              asian_popula...   76660 0.01
5029
#>  2 02    Alaska               asian_popula...   44032 0.05
6638
```

```
#>  3 04    Arizona              asian_popula...  257430 0.03
4746
#>  4 05    Arkansas             asian_popula...   51839 0.01
6922
#>  5 06    California           asian_popula... 6085947 0.13
3390
#>  6 08    Colorado             asian_popula...  199827 0.03
3452
#>  7 09    Connecticut          asian_popula...  172455 0.04
5642
#>  8 10    Delaware             asian_popula...   42699 0.04
1349
#>  9 11    District of Columbia asian_popula...   33545 0.04
6391
#> 10 12    Florida              asian_popula...  643682 0.02
9018
--snip--
```

This is a reasonable way to calculate the Asian population as a percentage of the total population in each state—but it's not the only way.

### *Using a Summary Variable*

Kyle Walker knew that calculating summaries like you've just done would be a common use case for `tidycensus`. To calculate, say, the Asian population as a percentage of the whole, you need to have a numerator (the Asian population) and denominator (the total population). So, to simplify things, Walker included the `summary_var` argument, which can be used within `get_decennial()` to import the total population as a separate variable. Instead of putting `P1_001N` (total population) in the `variables` argument and renaming it, you can assign it to the `summary_var` argument as follows:

```
get_decennial(
  geography = "state",
  variables = c(asian_population = "P1_006N"),
  summary_var = "P1_001N",
  year = 2020
) %>%
  arrange(NAME)
```

This returns a nearly identical data frame to what you just got, except that the total population is now a separate variable, rather than additional rows for each state:

```
#> # A tibble: 52 × 5
#>    GEOID NAME                 variable         value summa
r...
#>    <chr> <chr>                <chr>             <dbl>    <
dbl>
#>  1 01    Alabama              asian_popula...   76660  502
4279
#>  2 02    Alaska               asian_popula...   44032   73
3391
#>  3 04    Arizona              asian_popula...  257430  715
1502
#>  4 05    Arkansas             asian_popula...   51839  301
1524
#>  5 06    California           asian_popula... 6085947 3953
8223
#>  6 08    Colorado             asian_popula...  199827  577
3714
#>  7 09    Connecticut          asian_popula...  172455  360
5944
#>  8 10    Delaware             asian_popula...   42699   98
9948
#>  9 11    District of Columbia asian_popula...   33545   68
9545
#> 10 12    Florida              asian_popula...  643682 2153
8187
--snip--
#> #   summary_value
```

With the data in this new format, now you can calculate the Asian population as a percentage of the whole by dividing the `value` variable by the `summary_value` variable. Then you drop the `summary_value` variable because you no longer need it after doing this calculation:

```
get_decennial(
  geography = "state",
  variables = c(asian_population = "P1_006N"),
  summary_var = "P1_001N",
  year = 2020
) %>%
  arrange(NAME) %>%
  mutate(pct = value / summary_value) %>%
  select(-summary_value)
```

The resulting output is identical to the output of the previous section:

```
#> # A tibble: 52 × 5
#>    GEOID NAME                 variable          value
 pct
#>    <chr> <chr>                <chr>             <dbl>    <
dbl>
#>  1 01    Alabama              asian_popula...   76660 0.01
5258
#>  2 02    Alaska               asian_popula...   44032 0.06
0039
#>  3 04    Arizona              asian_popula...  257430 0.03
5997
#>  4 05    Arkansas             asian_popula...   51839 0.01
7214
#>  5 06    California           asian_popula... 6085947 0.15
3930
#>  6 08    Colorado             asian_popula...  199827 0.03
4610
#>  7 09    Connecticut          asian_popula...  172455 0.04
7825
#>  8 10    Delaware             asian_popula...   42699 0.04
3133
#>  9 11    District of Columbia asian_popula...   33545 0.04
8648
#> 10 12    Florida              asian_popula...  643682 0.02
9886
#> # 42 more rows
```

How you choose to calculate summary statistics is up to you; `tidycensus` makes it easy to do either way.

## Visualizing American Community Survey Data

Once you've accessed data using the `tidycensus` package, you can do whatever you want with it. In this section, you'll practice analyzing and visualizing survey data using the American Community Survey. This survey, which is conducted every year, differs from the Decennial Census in two major ways: it is given to a sample of people rather than the entire population, and it includes a wider range of questions.

Despite these differences, you can access data from the American Community Survey nearly identically to how you access Decennial Census data. Instead of `get_decennial()`, you use the `get_acs()` function, but the arguments you pass to it are the same:

```
get_acs(
  geography = "state",
  variables = "B01002_001",
  year = 2020
)
```

This code uses the `B01002_001` variable to get median age data from 2020 for each state. Here's what the output looks like:

```
#> # A tibble: 52 × 5
#>    GEOID NAME                 variable   estimate   moe
#>    <chr> <chr>                <chr>         <dbl> <dbl>
#>  1 01    Alabama              B01002_001     39.2   0.1
#>  2 02    Alaska               B01002_001     34.6   0.2
#>  3 04    Arizona              B01002_001     37.9   0.2
#>  4 05    Arkansas             B01002_001     38.3   0.2
#>  5 06    California           B01002_001     36.7   0.1
#>  6 08    Colorado             B01002_001     36.9   0.1
#>  7 09    Connecticut          B01002_001     41.1   0.2
#>  8 10    Delaware             B01002_001     41.0   0.2
#>  9 11    District of Columbia B01002_001     34.1   0.1
#> 10 12    Florida              B01002_001     42.2   0.2
--snip--
```

You should notice two differences in the output from `get_acs()` compared to that from `get_decennial()`. First, instead of the `value` column, `get_acs()` produces a column called `estimate`. Second, it adds a column called `moe`, for the margin of error. These changes are the result of American Community Survey being given only to a sample of the population, since extrapolating values from that sample to produce an estimate for the population as a whole introduces a margin of error.

In the state-level data, the margins of error are relatively low, but in smaller geographies, they tend to be higher. Cases in which your margins of error are high relative to your estimates indicate a greater level of uncertainty about how well the data represents the population as a whole, so you should interpret such results with caution.

### *Making Charts*

To pipe your data on median age into ggplot to create a bar chart, add the following lines:

```
get_acs(
  geography = "state",
  variables = "B01002_001",
  year = 2020
) %>%
  ggplot(aes(
    x = estimate,
    y = NAME
  )) +
  geom_col()
```

After importing the data with the `get_acs()` function, the `ggplot()` function pipes it directly into ggplot. States (which use the variable `NAME`) will go on the y-axis, and median age (`estimate`) will go on the x-axis. A simple `geom_col()` creates the bar chart shown in Figure 11-3.

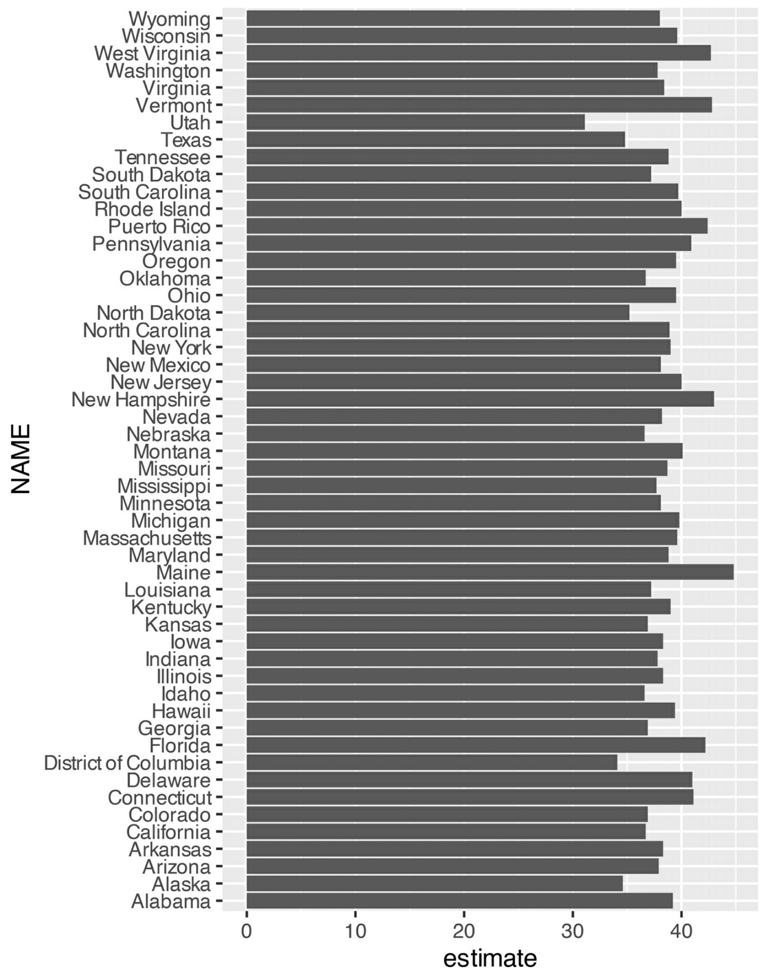


*Figure 11-3: A bar chart generated using data acquired with the get_asc() function*

This chart is nothing special, but the fact that it takes just six lines of code to create most definitely is!

## Making Population Maps with the geometry Argument

In addition to co-creating `tidycensus`, Kyle Walker created the `tigris` package for working with geospatial data. As a result, these packages are tightly integrated. Within the `get_acs()` function, you can set the `geometry` argument to `TRUE` to receive both demographic data from the Census Bureau and geospatial data from `tigris`:

```
get_acs(
  geography = "state",
  variables = "B01002_001",
  year = 2020,
  geometry = TRUE
)
```

In the resulting data, you can see that it has the metadata and `geometry` column of the simple features objects that you saw in Chapter 4:

```
#> Simple feature collection with 52 features and 5 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -179.1489 ymin: 17.88328 xmax: 179.77
85 ymax: 71.36516
#> Geodetic CRS:  NAD83
#> First 10 features:
#>    GEOID  NAME         variable   estimate  moe
#> 1  35     New Mexico   B01002_001     38.1  0.1
#> 2  72     Puerto Rico  B01002_001     42.4  0.2
#> 3  06     California   B01002_001     36.7  0.1
#> 4  01     Alabama      B01002_001     39.2  0.1
#> 5  13     Georgia      B01002_001     36.9  0.1
#> 6  05     Arkansas     B01002_001     38.3  0.2
#> 7  41     Oregon       B01002_001     39.5  0.1
#> 8  28     Mississippi  B01002_001     37.7  0.2
#> 9  08     Colorado     B01002_001     36.9  0.1
#> 10 49     Utah         B01002_001     31.1  0.1
#>    geometry
#> 1  MULTIPOLYGON (((-109.0502 3...
#> 2  MULTIPOLYGON (((-65.23805 1...
#> 3  MULTIPOLYGON (((-118.6044 3...
#> 4  MULTIPOLYGON (((-88.05338 3...
#> 5  MULTIPOLYGON (((-81.27939 3...
#> 6  MULTIPOLYGON (((-94.61792 3...
#> 7  MULTIPOLYGON (((-123.6647 4...
#> 8  MULTIPOLYGON (((-88.50297 3...
#> 9  MULTIPOLYGON (((-109.0603 3...
#> 10 MULTIPOLYGON (((-114.053 37...
```

The geometry type is `MULTIPOLYGON`, which you learned about in Chapter 4. To pipe this data into ggplot to make a map, add the following code:

```
get_acs(
  geography = "state",
  variables = "B01002_001",
  year = 2020,
  geometry = TRUE
) %>%
  ggplot(aes(fill = estimate)) +
  geom_sf() +
  scale_fill_viridis_c()
```

After importing the data with `get_acs()` and piping it into the `ggplot()` function, this code sets the `estimate` variable to use for the `fill` aesthetic property; that is, the fill color of each state will vary depending on the median age of its residents. Then `geom_sf()` draws the map, and the `scale_fill_viridis_c()` function gives it a colorblind-friendly palette.

The resulting map, shown in Figure 11-4, is less than ideal because the Aleutian Islands in Alaska cross the 180-degree line of longitude, or the International Date Line. As a result, most of Alaska appears on one side of the map and a small part appears on the other side. What's more, both Hawaii and Puerto Rico are hard to see.

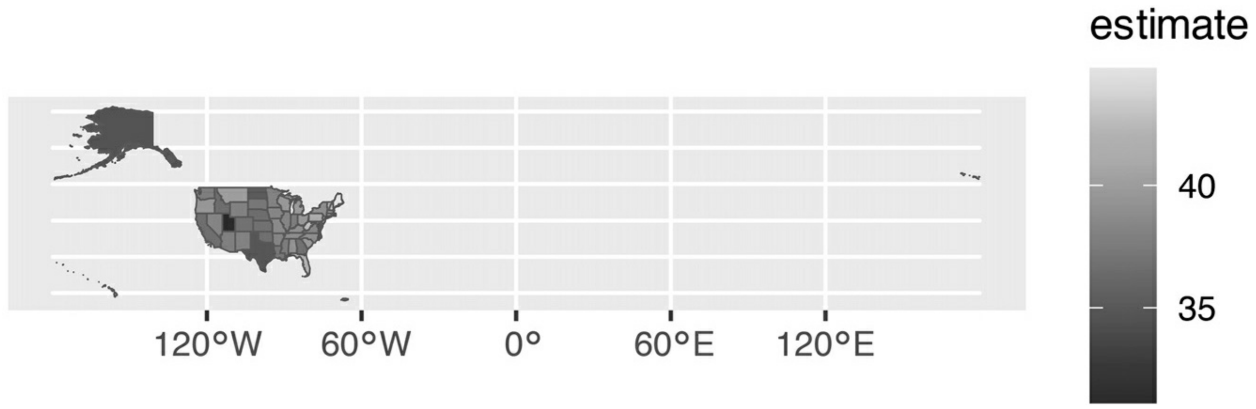


*Figure 11-4: A hard-to-read map showing median age by state*

To fix these problems, load the `tigris` package, then use the `shift_geometry()` function to move Alaska, Hawaii, and Puerto Rico into places where they'll be more easily visible:

```
library(tigris)

get_acs(
  geography = "state",
  variables = "B01002_001",
  year = 2020,
  geometry = TRUE
) %>%
  shift_geometry(preserve_area = FALSE) %>%
  ggplot(aes(fill = estimate)) +
  geom_sf() +
  scale_fill_viridis_c()
```

Setting the `preserve_area` argument to `FALSE` shrinks the giant state of Alaska and makes Hawaii and Puerto Rico larger. Although the state sizes in the map won't be precise, the map will be easier to read, as you can see in Figure 11-5.

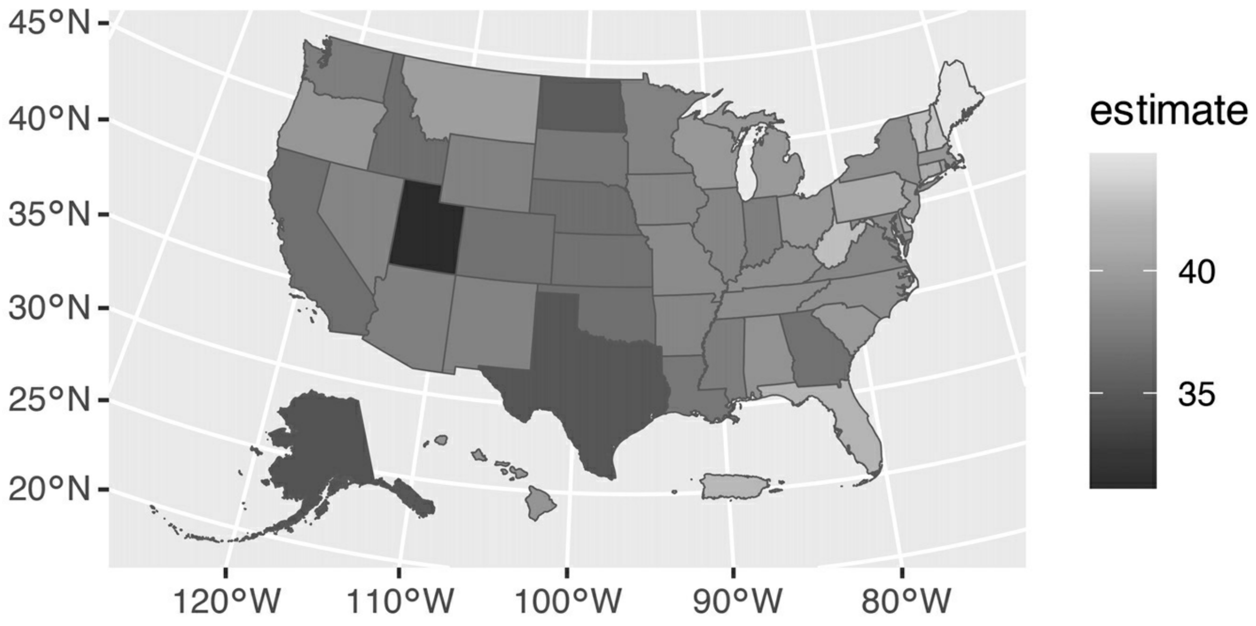


*Figure 11-5: An easier-to-read map tweaked using tigris functions*

Now try making the same map for all 3,000 counties by changing the `geography` argument to `"county"`. Other geographies include `region`, `tract` (for census tracts), `place` (for census-designated places, more commonly known as towns and cities), and `congressional district`. There are also many more arguments in both the `get_decennial()` and `get_acs()` functions; I've shown you only a few of the most common. If you want to learn more, Walker's book *Analyzing US Census Data: Methods, Maps, and*

*Models in R* is a great resource.

## Summary

This chapter explored two packages that use APIs to access data directly from its source. The `googlesheets4` package lets you import data from a Google Sheet. It's particularly useful when you're working with survey data, as it makes it easy to update your reports when new results come in. If you don't work with Google Sheets, you could use similar packages to fetch data from Excel365 (`Microsoft365R`), Qualtrics (`qualtRics`), Survey Monkey (`svmkrR`), and other sources.

If you work with US Census Bureau data, the `tidycensus` package is a huge time-saver. Rather than having to manually download data from the Census Bureau website, you can use `tidycensus` to write R code that accesses the data automatically, making it ready for analysis and reporting. Because of the package's integration with `tigris`, you can also easily map this demographic data.

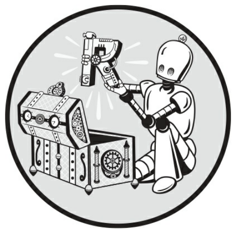
If you're looking for census data from other countries, there are also R packages to bring data from Canada (`cancensus`), Kenya (`rKenyaCensus`), Mexico (`mxmaps` and `inegiR`), Europe (`eurostat`), and other regions. Before hitting that download button in your data collection tool to get a CSV file, it's worth looking for a package that can import that data directly into R.

## Additional Resources

- Isabella Velásquez and Curtis Kephart, "Automated Survey Reporting with googlesheets4, pins, and R Markdown," Posit, June 15, 2022, *https://posit.co/blog/automated-survey-reporting/*.
- Kyle Walker, *Analyzing US Census Data: Methods, Maps, and Models in R* (Boca Raton, FL: CRC Press, 2023), *https://walker-data.com/census-r/*.

# 12

# CREATING FUNCTIONS AND PACKAGES



In this chapter, you will learn how to define your own R functions, including the parameters they should accept. Then, you'll create a package to distribute those functions, add your code and dependencies to it, write its documentation, and choose the license under which to release it.

Saving your code as custom functions and then distributing them in packages can have numerous benefits. First, packages make your code easier for others to use. For example, when researchers at the Moffitt Cancer Center needed to access code from a database, data scientists Travis Gerke and Garrick Aden-Buie used to write R code for each researcher, but they quickly realized they were reusing the same code over and over. Instead, they made a package with functions for accessing databases. Now researchers no longer had to ask for help; they could simply install the package Gerke and Aden-Buie had made and use its functions themselves.

What's more, developing packages allows you to shape how others work. Say you make a ggplot theme that follows the principles of high-quality data visualization discussed in Chapter 3. If you put this theme in a package, you can give others an easy way to follow these design principles. In short, functions and packages help you work with others using shared code.

## Creating Your Own Functions

Hadley Wickham, developer of the `tidyverse` set of packages, recommends creating a function once you've copied some code three times. Functions have three pieces: a name, a body, and arguments.

### *Writing a Simple Function*

You'll begin by writing an example of a relatively simple function. This function, called `show_in_excel_penguins()`, opens the penguin data from Chapter 7 in Microsoft Excel:

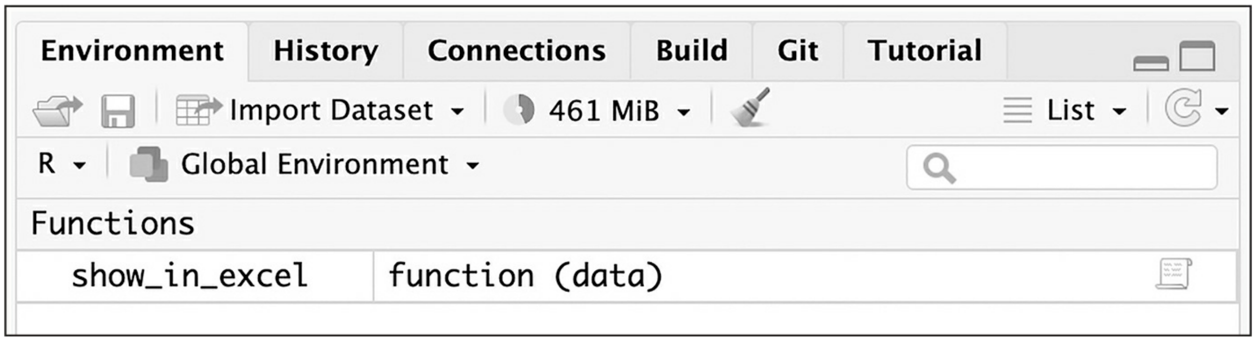
```
❶ penguins <- read_csv("https://data.rfortherestofus.com/pengui
ns-2007.csv")


❷ show_in_excel_penguins <- function() {
    csv_file <- str_glue("{tempfile()}.csv")

    write_csv(
      x = penguins,
      file = csv_file,
      na = ""
    )

    file_show(path = csv_file)
  }
```

This code first loads the `tidyverse` and `fs` packages. You'll use `tidyverse` to create a filename for the CSV file and save it, and `fs` to open the CSV file in Excel (or whichever program your computer uses to open CSV files by default).

Next, the `read_csv()` function imports the penguin data and names the data frame `penguins` ❶. Then it creates the new `show_in_excel_penguins` function, using the assignment operator (`<-`) and `function()` to specify that `show_in_excel_penguins` isn't a variable name but a function name ❷. The open curly bracket (`{`) at the end of the line indicates the start of the function body, where the "meat" of the function can be found. In this case, the body does three things:

- Creates a location for a CSV file to be saved using the `str_glue()` function combined with the `tempfile()` function. This creates a file at a temporary location with the *.csv* extension and saves it as `csv_file`.
- Writes `penguins` to the location set in `csv_file`. The `x` argument in `write_csv()` refers to the data frame to be saved. It also specifies that all `NA` values should show up as blanks. (By default, they would display the text *NA*.)
- Uses the `file_show()` function from the `fs` package to open the temporary CSV file in Excel.

To use the `show_in_excel_penguins()` function, highlight the lines that define the function and then press COMMAND-ENTER on macOS or CTRL-ENTER on Windows. You should now see the function in your global environment, as shown in Figure 12-1.



Functions

show_in_excel    function (data)

*Figure 12-1: The new function in the global environment*

From now on, any time you run the code `show_in_excel_penguins()`, R will open the `penguins` data frame in Excel.

## Adding Arguments

You're probably thinking that this function doesn't seem very useful. All it does is open the `penguins` data frame. Why would you want to keep doing that? A more practical function would let you open *any* data in Excel so you can use it in a variety of contexts.

The `show_in_excel()` function does just that: it takes any data frame from R, saves it as a CSV file, and opens the CSV file in Excel. Bruno

Rodrigues, head of the Department of Statistics and Data Strategy at the Ministry of Higher Education and Research in Luxembourg, wrote `show_in_excel()` to easily share data with his non-R-user colleagues. Whenever he needed data in a CSV file, he could run this function.

Replace your `show_in_excel_penguins()` function definition with this slightly simplified version of the code that Rodrigues used:

```
show_in_excel <- function(data) {
  csv_file <- str_glue("{tempfile()}.csv")
  write_csv(
    x = data,
    file = csv_file,
    na = ""
  )
  file_show(path = csv_file)
}
```

This code looks the same as `show_in_excel_penguins()`, with two exceptions. Notice that the first line now says `function(data)`. Items listed within the parentheses of the function definition are arguments. If you look farther down, you'll see the second change. Within `write_csv()`, instead of `x = penguins`, it now says `x = data`. This allows you to use the function with any data, not just `penguins`.

To use this function, you simply tell `show_in_excel()` what data to use, and the function opens the data in Excel. For example, tell it to open the `penguins` data frame as follows:

```
show_in_excel(data = penguins)
```

Having created the function with the `data` argument, now you can run it with any data you want to. This code, for example, imports the COVID case data from Chapter 10 and opens it in Excel:

```
covid_data <- read_csv("https://data.rfortherestofus.com/
us-states-covid-rolling-average.csv")
show_in_excel(data = covid_data)
```

You can also use `show_in_excel()` at the end of a pipeline. This code filters the `covid_data` data frame to include only data from California before opening it in Excel:

```
covid_data %>%
  filter(state == "California") %>%
  show_in_excel()
```

Rodrigues could have copied the code within the `show_in_excel()` function and rerun it every time he wanted to view his data in Excel. But, by creating a function, he was able to write the code just once and then run it as many times as necessary.

## *Creating a Function to Format Race and Ethnicity Data*

Hopefully now you better understand how functions work, so let's walk through an example function you could use to simplify some of the activities from previous chapters.

In Chapter 11, when you used the `tidycensus` package to automatically import data from the US Census Bureau, you learned that the census data has many variables with nonintuitive names. Say you regularly want to access data about race and ethnicity from the American Community Survey, but you can never remember which variables enable you to do so. To make your task more efficient, you'll create a `get_acs_race_ethnicity()` function step-by-step in this section, learning some important concepts about custom functions along the way.

A first version of the `get_acs_race_ethnicity()` function might look like this:

```
library(tidycensus)

get_acs_race_ethnicity <- function() {
  race_ethnicity_data <-
    get_acs(
      geography = "state",
      variables = c(
        "White" = "B03002_003",
        "Black/African American" = "B03002_004",
        "American Indian/Alaska Native" = "B03002_005",
        "Asian" = "B03002_006",
        "Native Hawaiian/Pacific Islander" = "B03002_007",
        "Other race" = "B03002_008",
        "Multi-Race" = "B03002_009",
        "Hispanic/Latino" = "B03002_012"
      )
    )

  race_ethnicity_data
}
```

Within the function body, this code calls the `get_acs()` function from `tidycensus` to retrieve population data at the state level. But instead of returning the function's default output, it updates the hard-to-remember variable names to human-readable names, such as `White` and `Black/African American`, and saves them as an object called `race_ethnicity_data`. The code then uses the `race_ethnicity_data` object to return that data when the `get_acs_race_ethnicity()` function is run.

To run this function, enter the following:

```
get_acs_race_ethnicity()
```

Doing so should return data with easy-to-read race and ethnicity group names:

```
#> # A tibble: 416 × 5
#>    GEOID NAME    variable                         estimate   m
oe
#>    <chr> <chr>   <chr>                               <dbl> <db
l>
#>  1 01    Alabama White                             3241003  20
76
#>  2 01    Alabama Black/African American            1316314  30
18
#>  3 01    Alabama American Indian/Alaska Na...        17417   9
41
#>  4 01    Alabama Asian                               69331  15
59
#>  5 01    Alabama Native Hawaiian/Pacific I...         1594   3
76
#>  6 01    Alabama Other race                          12504  18
```

```
67
#>  7 01    Alabama Multi-Race                       114853  38
35
#>  8 01    Alabama Hispanic/Latino                  224659   4
13
#>  9 02    Alaska  White                            434515  10
67
#> 10 02    Alaska  Black/African American            22787   7
69
#> # 406 more rows
```

You could improve this function in a few ways. You might want the resulting variable names to follow a consistent syntax, for example, so you could use the `clean_names()` function from the `janitor` package to format them in *snake case* (in which all words are lowercase and separated by underscores). However, you might also want to have the option of keeping the original variable names. To accomplish this, add the `clean_variable_names` argument to the function definition as follows:

```
get_acs_race_ethnicity <- function(clean_variable_names = FAL
SE) {
  race_ethnicity_data <-
    get_acs(
      geography = "state",
      variables = c(
        "White" = "B03002_003",
        "Black/African American" = "B03002_004",
        "American Indian/Alaska Native" = "B03002_005",
        "Asian" = "B03002_006",
        "Native Hawaiian/Pacific Islander" = "B03002_007",
        "Other race" = "B03002_008",
        "Multi-Race" = "B03002_009",
        "Hispanic/Latino" = "B03002_012"
      )
    )

  if (clean_variable_names == TRUE) {
 ❶ race_ethnicity_data <- clean_names(race_ethnicity_data)
  }

  race_ethnicity_data
}
```

This code adds the `clean_variable_names` argument to `get_acs_race_ethnicity()` and specifies that its value should be `FALSE` by default. Then, in the function body, an `if` statement says that if the argument is `TRUE`, the variable names should be overwritten by versions formatted in snake case ❶. If the argument is `FALSE`, the variable names remain unchanged.

If you run the function now, nothing should change, because the new argument is set to `FALSE` by default. Try setting `clean_variable_names` to `TRUE` as follows:

```
get_acs_race_ethnicity(clean_variable_names = TRUE)
```

This function call should return data with consistent variable names:

```
#> # A tibble: 416 × 5
#>    geoid name    variable                      estimate   m
oe
#>    <chr> <chr>   <chr>                            <dbl> <db
l>
#>  1 01    Alabama White                          3241003  20
76
#>  2 01    Alabama Black/African American         1316314  30
18
#>  3 01    Alabama American Indian/Alaska Na...     17417   9
41
#>  4 01    Alabama Asian                            69331  15
59
#>  5 01    Alabama Native Hawaiian/Pacific I...      1594   3
76
#>  6 01    Alabama Other race                       12504  18
67
#>  7 01    Alabama Multi-Race                      114853  38
35
#>  8 01    Alabama Hispanic/Latino                 224659   4
13
#>  9 02    Alaska  White                           434515  10
67
#> 10 02    Alaska  Black/African American           22787   7
69
#> # 406 more rows
```

Notice that `GEOID` and `NAME` now appear as `geoid` and `name`.

Now that you've seen how to add arguments to two separate functions, you'll learn how to pass arguments from one function to another.

## *Using ... to Pass Arguments to Another Function*

The `get_acs_race_ethnicity()` function you've created retrieves population data at the state level by passing the `geography = "state"` argument to the `get_acs()` function. But what if you wanted to obtain county-level or census tract data? You could do so using `get_acs()`, but `get_acs_race_ethnicity()` isn't currently written in a way that would allow this. How could you modify the function to make it more flexible?

Your first idea might be to add a new argument for the level of data to retrieve. You could edit the first two lines of the function as follows to add a `my_geography` argument and then use it in the `get_acs()` function like so:

```
get_acs_race_ethnicity <- function(clean_variable_names = FAL
SE, my_geography) {
  race_ethnicity_data <- get_acs(geography = my_geography,
--snip--
```

But what if you also want to select the year for which to retrieve data? Well, you could add an argument for that as well. However, as you saw in Chapter 11, the `get_acs()` function has many arguments, and repeating them all in your code would quickly become cumbersome.

The `...` syntax gives you a more efficient option. Placing `...` in the `get_acs_race_ethnicity()` function allows you to automatically pass any of its arguments to `get_acs()` by including `...` in that function as well:

```
get_acs_race_ethnicity <- function(
  clean_variable_names = FALSE,
  ...
) {
  race_ethnicity_data <-
    get_acs(
      ...,
      variables = c(
        "White" = "B03002_003",
        "Black/African American" = "B03002_004",
        "American Indian/Alaska Native" = "B03002_005",
        "Asian" = "B03002_006",
        "Native Hawaiian/Pacific Islander" = "B03002_007",
        "Other race" = "B03002_008",
        "Multi-Race" = "B03002_009",
        "Hispanic/Latino" = "B03002_012"
      )
    )

  if (clean_variable_names == TRUE) {
    race_ethnicity_data <- clean_names(race_ethnicity_data)
  }

  race_ethnicity_data
}
```

Try running your function by passing it the `geography` argument set to `"state"`:

```
get_acs_race_ethnicity(geography = "state")
```

This should return the following:

```
#> # A tibble: 416 × 5
#>    GEOID NAME    variable                         estimate   m
oe
#>    <chr> <chr>   <chr>                               <dbl> <db
l>
#>  1 01    Alabama White                             3241003  20
76
#>  2 01    Alabama Black/African American            1316314  30
18
#>  3 01    Alabama American Indian/Alaska Na...        17417   9
41
#>  4 01    Alabama Asian                               69331  15
59
#>  5 01    Alabama Native Hawaiian/Pacific I...         1594   3
76
#>  6 01    Alabama Other race                          12504  18
67
#>  7 01    Alabama Multi-Race                         114853  38
35
#>  8 01    Alabama Hispanic/Latino                    224659   4
```

```
13
#>  9 02    Alaska  White                               434515  10
67
#> 10 02    Alaska  Black/African American               22787   7
69
#> # 406 more rows
```

You'll see that the `GEOID` and `NAME` variables are uppercase because the `clean_variable_names` argument is set to `FALSE` by default, and we didn't change it when using the `get_acs_race_ethnicity()` function.

Alternatively, you could change the value of the argument to get data by county:

```
get_acs_race_ethnicity(geography = "county")
```

You could also run the function with the `geometry = TRUE` argument to return geospatial data alongside demographic data:

```
get_acs_race_ethnicity(geography = "county", geometry = TRUE)
```

The function should return data like the following:

```
#> Simple feature collection with 416 features and 5 fields
#> Geometry type: MULTIPOLYGON
#> Dimension:     XY
#> Bounding box:  xmin: -179.1489 ymin: 17.88328 xmax: 179.77
85 ymax: 71.36516
#> Geodetic CRS:  NAD83
#> First 10 features:
#>    GEOID NAME    variable                          estimate
#>  1 56    Wyoming White                               478508
#>  2 56    Wyoming Black/African American                4811
#>  3 56    Wyoming American Indian/Alaska Na...  11330
#>  4 56    Wyoming Asian                                 4907
#>  5 56    Wyoming Native Hawaiian/Pacific I...    397
#>  6 56    Wyoming Other race                            1582
#>  7 56    Wyoming Multi-Race                           15921
#>  8 56    Wyoming Hispanic/Latino                      59185
#>  9 02    Alaska  White                               434515
#> 10 02    Alaska  Black/African American               22787
#>     moe geometry
#> 1   959 MULTIPOLYGON (((-111.0546 4...
#> 2   544 MULTIPOLYGON (((-111.0546 4...
#> 3   458 MULTIPOLYGON (((-111.0546 4...
#> 4   409 MULTIPOLYGON (((-111.0546 4...
#> 5   158 MULTIPOLYGON (((-111.0546 4...
#> 6   545 MULTIPOLYGON (((-111.0546 4...
#> 7  1098 MULTIPOLYGON (((-111.0546 4...
#> 8   167 MULTIPOLYGON (((-111.0546 4...
#> 9  1067 MULTIPOLYGON (((179.4825 51...
#> 10  769 MULTIPOLYGON (((179.4825 51...
```

The `...` syntax allows you to create your own function and pass arguments from it to another function without repeating all of that function's arguments in your own code. This approach gives you flexibility while keeping your code concise.

Now let's look at how to put your custom functions into a package.

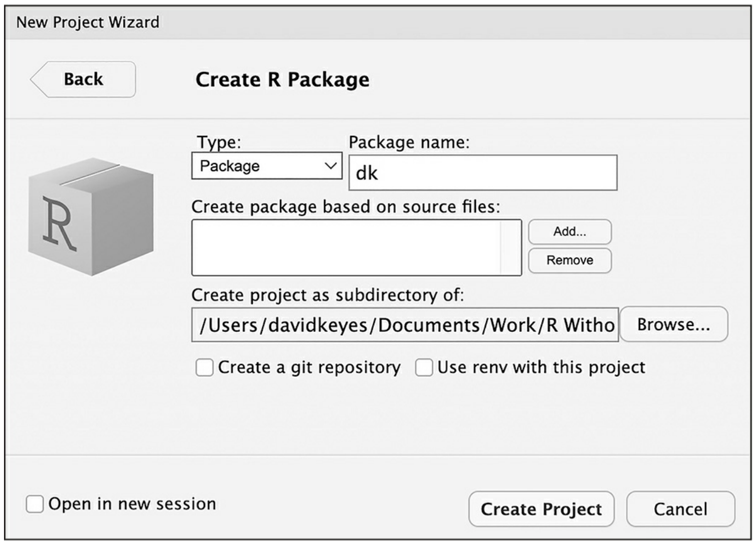
## Creating a Package

Packages bundle your functions so you can use them in multiple projects. If you find yourself copying functions from one project to another, or from a *functions.R* file into each new project, that's a good indication that you should make a package.

While you can run the functions from a *functions.R* file in your own environment, this code might not work on someone else's computer. Other users may not have the necessary packages installed, or they may be confused about how your functions' arguments work and not know where to go for help. Putting your functions in a package makes them more likely to work for everyone, as they include the necessary dependencies as well as built-in documentation to help others use the functions on their own.

### *Starting the Package*

To create a package in RStudio, go to **File ▸ New Project ▸ New Directory**. Select **R Package** from the list of options and give your package a name. In Figure 12-2, I've called mine `dk`. Also decide where you want your package to live on your computer. You can leave everything else as is.



Type: Package
Package name: dk
Create package based on source files:
Create project as subdirectory of: /Users/davidkeyes/Documents/Work/R Witho
☐ Create a git repository ☐ Use renv with this project
☐ Open in new session

*Figure 12-2: The RStudio menu for creating your own package*

RStudio will now create and open the package. It should already contain a few files, including *hello.R*, which has a prebuilt function called `hello()` that, when run, prints the text `Hello, world!` in the console. You'll get rid of this and a few other default files so you can start with a clean slate. Delete *hello.R*, *NAMESPACE*, and *hello.Rd* in the *man* directory.

### Adding Functions with use_r()

All of the functions in a package should go in separate files in the *R* folder. To add these files to the package automatically and test that they work correctly, you'll use the `usethis` and `devtools` packages. Install them using `install.packages()` like so:

```
install.packages("usethis")
install.packages("devtools")
```

To add a function to the package, run the `use_r()` function from the `usethis` package in the console:

```
usethis::use_r("acs")
```

The *`package::function()`* syntax allows you to use a function without loading the associated package. The `use_r()` function should create a file in the *R* directory with the argument name you provide—in this case, the file is called *acs.R*. The name itself doesn't really matter, but it's a good practice to choose something that gives an indication of the functions the file contains. Now you can open the file and add code to it. Copy the `get_acs_race_ethnicity()` function to the package.

### *Checking the Package with devtools*

You need to change the `get_acs_race_ethnicity()` function in a few ways to make it work in a package. The easiest way to figure out what changes you need to make is to use built-in tools to check that your package is built correctly. Run the function **`devtools::check()`** in the console to perform what is known as an `R CMD check`, a command that runs under the hood to ensure others can install your package on their system. Running `R CMD check` on the `dk` package outputs this long message:

```
── R CMD check results ─────────────── dk ────
Duration: 4s

> checking DESCRIPTION meta-information ... WARNING
Non-standard license specification:
What license is it under?
Standardizable: FALSE

> checking for missing documentation entries ... WARNING
Undocumented code objects:
'get_acs_race_ethnicity'
All user-level objects in a package should have documentation
 entries.
See chapter 'Writing R documentation files' in the 'Writing R
Extensions' manual.


❶ > checking R code for possible problems ... NOTE
get_acs_race_ethnicity: no visible global function definition
 for
'get_acs'
get_acs_race_ethnicity: no visible global function definition
 for
'clean_names'
Undefined global functions or variables:
```

```
clean_names get_acs


0 errors ✔ | 2 warnings x | 1 note x
```

The last part is the most important, so let's review the output from bottom to top. The line `0 errors ✔ | 2 warnings x | 1 note x` highlights three levels of issues identified in the package. Errors are the most severe, as they mean others won't be able to install your package, while warnings and notes may cause problems for others. It's best practice to eliminate all errors, warnings, and notes.

We'll start by addressing the note at ❶. To help you understand what `R CMD check` is saying here, I need to explain a bit about how packages work. When you install a package using the `install.packages()` function, it often takes a while. That's because the package you're telling R to install likely uses functions from other packages. To access these functions, R must install these packages (known as *dependencies*) for you; after all, it would be a pain if you had to manually install a whole set of dependencies every time you installed a new package. But to make sure that the appropriate packages are installed for any user of the `dk` package, you still have to make a few changes.

`R CMD check` is saying this package includes several "undefined global functions or variables" and "no visible global function definition" for various functions. This is because you're trying to use functions from the `tidycensus` and `janitor` packages, but you haven't specified where these functions come from. I can run this code in my environment because I have `tidycensus` and `janitor` installed, but you can't assume the same of everyone.

### Adding Dependency Packages

To ensure the package's code will work, you need to install `tidycensus` and `janitor` for users when they install the `dk` package. To do this, run the `use_package()` function from the `usethis` package in the console, first specifying `"tidycensus"` for the `package` argument:

```
usethis::use_package(package = "tidycensus")
```

You should get the following message:

```
✔ Setting active project to '/Users/davidkeyes/Documents/Wor
k/R for the Rest of Us/dk'
✔ Adding 'tidycensus' to Imports field in DESCRIPTION
• Refer to functions with `tidycensus::fun()`
```

The `Setting active project...` line indicates that you're working in the `dk` project. The second line indicates that the *DESCRIPTION* file has been edited. This file provides metadata about the package you're developing.

Next, add the `janitor` package the same way you added `tidyverse`

```
usethis::use_package(package = "janitor")
```

which should give you the following output:

```
✔ Adding 'janitor' to Imports field in DESCRIPTION
• Refer to functions with `janitor::fun()`
```

If you open the *DESCRIPTION* file in the root directory of your project, you should see the following:

```
Package: dk
Type: Package
Title: What the Package Does (Title Case)
Version: 0.1.0
Author: Who wrote it
Maintainer: The package maintainer <yourself@somewhere.net>
Description: More about what it does (maybe more than one lin
e)
    Use four spaces when indenting paragraphs within the Desc
ription.
License: What license is it under?
Encoding: UTF-8
LazyData: true
Imports:
    janitor,
    tidycensus
```

The `Imports` section at the bottom of the file indicates that when a user installs the `dk` package, the `tidycensus` and `janitor` packages will also be imported.

### *Referring to Functions Correctly*

The output from running `usethis::use_package(package = "janitor")` also included the line `Refer to functions with tidycensus::fun()` (where `fun()` stands for function name). This tells you that in order to use functions from other packages in the `dk` package, you need to specify both the package name and the function name to ensure that the correct function is used at all times. On rare occasions, you'll find functions with identical names used across multiple packages, and this syntax avoids ambiguity. Remember this line from the `R CMD check`?

```
Undefined global functions or variables:
clean_names get_acs
```

It appeared because you were using functions without saying what package they came from. The `clean_names()` function comes from the `janitor` package, and `get_acs()` comes from `tidycensus`, so you will need to add these package names before each function:

```
get_acs_race_ethnicity <- function(
  clean_variable_names = FALSE,
  ...
) {
  race_ethnicity_data <- tidycensus::get_acs(
    ...,
    variables = c(
      "White" = "B03002_003",
      "Black/African American" = "B03002_004",
      "American Indian/Alaska Native" = "B03002_005",
      "Asian" = "B03002_006",
      "Native Hawaiian/Pacific Islander" = "B03002_007",
      "Other race" = "B03002_008",
      "Multi-Race" = "B03002_009",
      "Hispanic/Latino" = "B03002_012"
    )
  )
  if (clean_variable_names == TRUE) {
    race_ethnicity_data <- janitor::clean_names(race_ethnicit
y_data)
  }

  race_ethnicity_data
}
```

Now you can run `devtools::check()` again, and you should see that the notes have gone away:

```
> checking DESCRIPTION meta-information ... WARNING
Non-standard license specification:
What license is it under?
Standardizable: FALSE

> checking for missing documentation entries ... WARNING
Undocumented code objects:
'get_acs_race_ethnicity'
All user-level objects in a package should have documentation
 entries.
See chapter 'Writing R documentation files' in the 'Writing R
Extensions' manual.

0 errors ✔ | 2 warnings x | 0 notes ✔
```

However, there are still two warnings to deal with. You’ll do that next.

### *Creating Documentation with Roxygen*

The `checking for missing documentation entries` warning indicates that you need to document your `get_acs_race_ethnicity()` function. One of the benefits of creating a package is that you can add documentation to help others use your code. In the same way that users can enter `?get_acs()` and see documentation about that function, you want them to be able to enter `?get_acs_race_ethnicity()` to learn how your function works.

To create documentation for `get_acs_race_ethnicity()`, you’ll use Roxygen, a documentation tool that uses a package called `roxygen2`. To get started, place your cursor anywhere in your function. Then, in RStudio go to **Code ▸ Insert Roxygen Skeleton**. This should add the following text before the `get _acs_race_ethnicity()` function:

```
#' Title
#'
#' @param clean_variable_names
#' @param ...
#'
#' @return
#' @export
#'
#' @examples
```

This text is the documentation's skeleton. Each line starts with the special characters `#'`, which indicate that you're working with Roxygen. Now you can edit the text to create your documentation. Begin by replacing `Title` with a sentence that describes the function:

```
#' Access race and ethnicity data from the American Community
 Survey
```

Next, turn your attention to the lines beginning with `@param`. Roxygen automatically creates one of these lines for each function argument, but it's up to you to fill them in with a description. Begin by describing what the `clean_variable_names` argument does. Next, specify that the `...` will pass additional arguments to the `tidycensus::get_acs()` function:

```
#' @param clean_variable_names Should variable names be clean
ed (i.e. snake case)
#' @param ... Other arguments passed to tidycensus::get_acs()
```

The `@return` line should tell the user what the `get_acs_race_ethnicity()` function returns. In this case, it returns data, which you document as follows:

```
#' @return A tibble with five variables: GEOID, NAME, variabl
e, estimate, and moe
```

After `@return` is `@export`. You don't need to change anything here. Most functions in a package are known as *exported functions*, meaning they're available to users of the package. In contrast, internal functions, which are used only by the package developers, don't have `@export` in the Roxygen skeleton.

The last section is `@examples`. This is where you can give examples of code that users can run to learn how the function works. Doing this introduces some complexity and isn't required, so you can skip it here and delete the line with `@examples` on it.

**NOTE**

*If you want to learn more about adding examples to your documentation, the second edition of Hadley Wickham and Jenny Bryan's book* R Packages *is a great resource.*

Now that you've added documentation with Roxygen, run **`devtools::document()`** in the console. This should create a *get_acs_race_ethnicity.Rd* documentation file in the *man* directory using the very specific format that R packages require. You're welcome to look at it, but you can't change it; it's read-only.

Running the function should also create a *NAMESPACE* file, which lists the functions that your package makes available to users. It should look like this:

```
# Generated by roxygen2: do not edit by hand

export(get_acs_race_ethnicity)
```

Your `get_acs_race_ethnicity()` function is now almost ready for users.

## *Adding a License and Metadata*

Run **`devtools::check()`** again to see if you've fixed the issues that led to the warnings. The warning about missing documentation should no longer be there. However, you do still get one warning:

```
> checking DESCRIPTION meta-information ... WARNING
```

```
Non-standard license specification:
What license is it under?
Standardizable: FALSE

0 errors ✔ | 1 warning x | 0 notes ✔
```

This warning reminds you that you have not given your package a license. If you plan to make your package publicly available, choosing a license is important because it tells other people what they can and cannot do with your code. For information about how to choose the right license for your package, see *https://choosealicense.com*.

In this example, you'll use the MIT license, which allows users to do essentially whatever they want with your code, by running **`usethis::use_mit_license()`**. The `usethis` package has similar functions for other common licenses. You should get the following output:

```
✔ Setting active project to '/Users/davidkeyes/Documents/Wor
k/R for the Rest of Us/dk'
✔ Setting License field in DESCRIPTION to 'MIT + file LICENS
E'
✔ Writing 'LICENSE'
✔ Writing 'LICENSE.md'
✔ Adding '^LICENSE\\.md$' to '.Rbuildignore'
```

The `use_mit_license()` function handles a lot of the tedious parts of adding a license to your package. Most importantly for our purposes, it specifies the license in the *DESCRIPTION* file. If you open it, you should see this confirmation that you've added the MIT license:

```
License: MIT + file LICENSE
```

In addition to the license, the *DESCRIPTION* file contains metadata about the package. You can make a few changes to identify its title and add an author, a maintainer, and a description. The final *DESCRIPTION* file might look something like this:

```
Package: dk
```

```
Type: Package
Title: David Keyes's Personal Package
Version: 0.1.0
Author: David Keyes
Maintainer: David Keyes <david@rfortherestofus.com>
    Description: A package with functions that David Keyes ma
y find
    useful.
License: MIT + file LICENSE
Encoding: UTF-8
LazyData: true
Imports:
    janitor,
    tidycensus
```

Having made these changes, run **`devtools::check()`** one more time to make sure everything is in order:

```
0 errors ✔ | 0 warnings ✔ | 0 notes ✔
```

This is exactly what you want to see!

### *Writing Additional Functions*

You've now got a package with one working function in it. If you wanted to add more functions, you would follow the same procedure:

1. Create a new *.R* file with `usethis::use_r()` or copy another function to the existing *.R* file.
2. Develop your function using the `package::function()` syntax to refer to functions from other packages.
3. Add any dependency packages with `use_package()`.
4. Add documentation for your function.
5. Run `devtools::check()` to make sure you did everything correctly.

Your package can contain a single function, like `dk`, or as many functions as you want.

### *Installing the Package*

Now you're ready to install and use the new package. When you're developing your own package, installing it for your own use is relatively straightforward. Simply run **`devtools::install()`**, and the package will be ready for you to use in any project.

Of course, if you're developing a package, you're likely doing it not just for yourself but for others as well. The most common way to make your package available to others is with the code-sharing website GitHub. The details of how to put your code on GitHub are beyond what I can cover here, but the book *Happy Git and GitHub for the useR* by Jenny Bryan (self-published at *https://happygitwithr.com*) is a great place to start.

I've pushed the `dk` package to GitHub, and you can find it at *https://github.com/dgkeyes/dk*. If you'd like to install it, first make sure you have the `remotes` package installed, then run the code **`remotes::install_github("dgkeyes/dk")`** in the console.

## Summary

In this chapter, you saw that packages are useful because they let you bundle several elements needed to reliably run your code: a set of functions, instructions to automatically install dependency packages, and code documentation.

Creating your own R package is especially beneficial when you're working for an organization, as packages can allow advanced R users to help colleagues with less experience. When Travis Gerke and Garrick Aden-Buie provided researchers at the Moffitt Cancer Center with a package that contained functions for easily accessing their databases, the researchers began to use R more creatively.

If you create a package, you can also guide people to use R in the way you think is best. Packages are a way to ensure that others follow best practices (without even being aware they are doing so). They make it easy to reuse functions across projects, help others, and adhere to a consistent style.

## Additional Resources

- Malcolm Barrett, "Package Development with R," online course, accessed December 2, 2023, *https://rfortherestofus.com/courses/package-*

*development*.

- Hadley Wickham and Jennifer Bryan, *R Packages*, 2nd ed. (Sebastopol, CA: O’Reilly Media, 2023), *https://r-pkgs.org*.

## Wrapping Up

R was invented in 1993 as a tool for statistics, and in the years since, it has been used for plenty of statistical analysis. But over the last three decades, R has also become a tool that can do much more than statistics.

As you’ve seen in this book, R is great for making visualizations. You can use it to create high-quality graphics and maps, make your own theme to keep your visuals consistent and on-brand, and generate tables that look good and communicate well. Using R Markdown or Quarto, you can create reports, presentations, and websites. And best of all, these documents are all reproducible, meaning that updating them is as easy as rerunning your code. Finally, you’ve seen that R can help you automate how you access data, as well as assist you in collaborating with others through the functions and packages you create.

If R was new to you when you started this book, I hope you now feel inspired to use it. If you’re an experienced R user, I hope this book has shown you some ways to use R that you hadn’t previously considered. No matter your background, my hope is that you now understand how to use R like a pro. Because it isn’t just a tool for statisticians—R is a tool for the rest of us too.

# INDEX