R Without Statistics

David Keyes

Contents

About the Book

This is the in-progress version of *R Without Statistics*, a forthcoming book from No Starch Press.

Since R was invented in 1993, it has become a widely used programming language for statistical analysis. From academia to the tech world and beyond, R is used for a wide range of statistical analysis.

R's ubiquity in the world of statistics leads many to assume that it is only useful to those who do complex statistical work. But as R has grown in popularity, the number of ways it can be used has grown as well. Today, R is used for:

- Data visualization
- Map making
- Sharing results through reports, slides, and websites
- Automating processes
- And much more!

The idea that R is only for statistical analysis is outdated and inaccurate. But, without a single book that demonstrates the power of R for non-statistical purposes, this perception persists.

Enter R Without Statistics.

R Without Statistics will show ways that R can be used beyond complex statistical analysis. Readers will learn about a range of uses for R, many of which they have likely never even considered.

Each chapter will, using a consistent format, cover one novel way of using R.

1. Readers will first be introduced to an R user who has done something novel and learn how using R in this way transformed their work.
2. Following this, there will be code samples that demonstrate exactly how the R user did the thing they are being profiled for.
3. Finally, there will be a summary, with lessons learned from this novel way of using R.

Written by David Keyes, Founder and CEO of R for the Rest of Us, R Without Statistics will be published by No Starch Press.

Introduction

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 of these was the R programming language (yes, really).

How did a humble tool for data analysis help New Zealand fight COVID-19? It allowed a team at the Ministry of Health to generate daily reports on cases throughout New Zealand. These reports enabled officials to develop policies that kept the country largely COVID-19 free. The team was small, however, so producing these 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 re-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 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 activities like the following:

- Make professional-quality data visualizations, maps, and tables.
- Replace a clunky multi-tool workflow for creating 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 a Tool for Statistical Analysis?

Many people think of R as simply a tool for hardcore statistical analysis. But, over a quarter of a century since its creation, R can do much more than manipulate numerical values. After all, every R user must illuminate their findings and communicate their results somehow, whether via data visualizations, reports, websites, or presentations. Also, the more you use R, the more you'll find yourself wanting to automate tasks you used to 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 R is good at these things. 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 replace other tools, R can allow you to do things, like generate reports and tables, that you're already probably doing, and it can do it 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 are either a current R user keen to explore uses of R for visualization and communication or a non-R user wondering if R is right for you. I've written *R Without Statistics* so that it should make sense even if you've never written a line of R code. But if you have written many lines of R code, the book should help you learn plenty of new techniques to up your R 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 we cover. We'll dive into their code, breaking the programs down to help you understand how they works, and suggest ways of going beyond the example. The book has three parts:

Part 1: Visualizations

In the first part, you'll learn about ways to use R to visualize data.

Chapter 1: An R Programming Crash Course

Introduces you to the R Studio 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, it introduces you to the `ggplot2` package for data visualization and addresses important principles that can help you to make high-quality graphics.

Chapter 3: Making Your Own ggplot Theme

Describes how journalists at the BBC made a custom theme for the `ggplot2` data visualization package. We'll walk through the package they created, and in the process, you'll learn how to make your own theme.

Chapter 4: Creating Maps

Walks through 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: Crafting High-Quality Tables

Shows you how to use the `gt` package to make high-quality tables in R. We draw from a conversation with R table connoisseur Tom Mock to learn how to apply design principles that ensure your tables communicate effectively.

Part 2: Reports, Presentations, and Websites

The second part of the book focuses on using R Markdown to communicate efficiently. You'll learn how to incorporate visualizations like the ones discussed in Part 1 into complete reports, slideshow presentations, and static websites generated entirely using R code.

Chapter 6: Writing Reports in R Markdown

Introduces R Markdown, a tool that allows you to generate a professional report in R. This chapter will cover the structure of an R Markdown document, show 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: Using Parameters to Automate Reports

Covers one of the advantages of using R Markdown: the fact that you can produce multiple reports at the same time using a technique called parameterized reporting. We explore 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: Making Slideshow Presentations with xaringan

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: Building Websites with distill

Shows you how to create your own website with R Markdown and the `distill` package. We explore how `distill` works by considering a website about COVID-19 rates in Westchester County, New York. In the process, we cover how to create pages on your site, add interactivity through R packages, and deploy your website using several options.

Chapter 10: Reproducible Reporting with Quarto

Explains how to use Quarto, the next-generation version of R Markdown. You'll learn to use Quarto to do all of the things you did previously in R Markdown (reports, parameterized reporting, slideshow presentations, and websites).

Part 3: Automation and Collaboration

The last part of the book focuses on ways you can use R to automate your work and share it with others.

Chapter 10: 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 United States Census Bureau data. You'll learn how the packages work and how to use them to automate the process of accessing data.

Chapter 11: Creating Your Own R Packages

Shows you how to create your own functions and packages to share your code with others. One of the major benefits of R is that you can create your own functions to automate common tasks, then bundle them into a package other users can import. This chapter covers a few example functions. Then you'll learn how to create your own package by learning from R developers who built packages to improve the work of researchers at the Moffitt Cancer Center.

By the end of the book, you should be able to use R for a wide range of non-statistical tasks. You'll know 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 yourself can develop. Let's dive in.

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 those who have never used R before. You'll set up an R programming environment with RStudio and learn how to work with data using functions, objects, packages, and projects. You'll also be introduced to the `tidyverse` package, which contains the core data analysis and manipulation functions we'll use in this book.

If you have prior experience with R, feel free to skip this chapter, but if you're just starting out, it should help you make sense of the rest of the book.

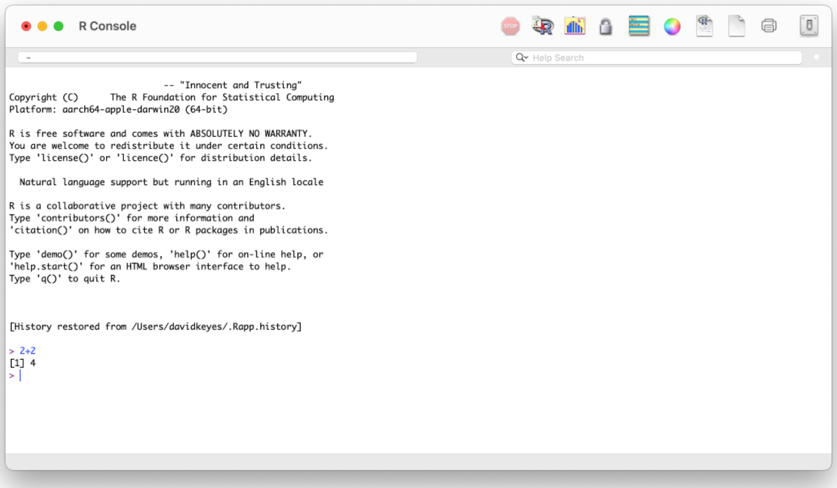
Setting Up

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 the book *Modern Dive* by Chester Ismay and Albert Kim: R is the engine that powers your data; RStudio is like a 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 in Figure **??** that lets you work with R on your operating system's command line. For example, enter `2 + 2`, and you should see `4`.

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.



```
                              -- "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: The R console

Exploring the RStudio Interface

The first time you open RStudio, you should see the three panels shown in Figure **??**.

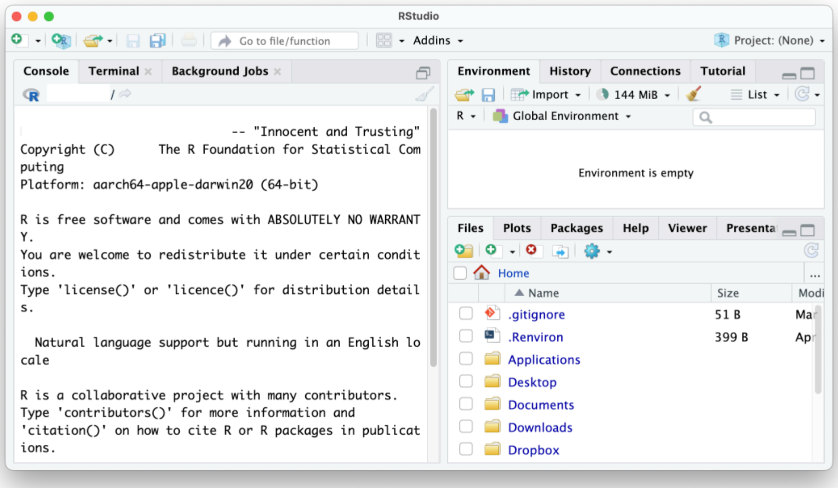
The left panel should look familiar. It's similar to the screen you saw when working in R on the command line. This is known as the *console*. You'll use it to enter code and see the results. This panel, like the others we'll discuss, has several tabs, such as Terminal and Background Jobs, for more advanced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

At the bottom right, the *files* panel shows all of the files on your computer. You can click any file to open it within RStudio. Finally, the top-right panel shows your *environment*, or the objects that are available to you when working in RStudio. We discuss objects below.

There is one more panel that you'll typically use when working in RStudio, but to make it appear, 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 recreate that code from scratch tomorrow. Writing your code in files lets you run it multiple times. There are two types of files we'll discuss in this book:



```
                          -- "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.
```

Environment is empty

Figure 2: The RStudio editor

- R script files, which contain only code.
- R Markdown files, which contain both code and text.

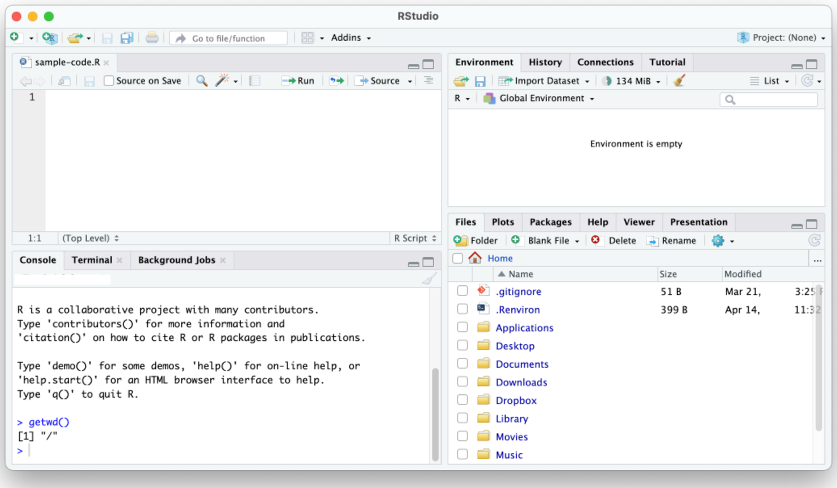
We'll talk about R Markdown files starting in Chapter **??**. For now, let's work with R script files, which use the *.R* extension. To create an R script file, go to File > New File > R Script. When you create a new R script file, a fourth panel should appear in the top left of R Studio, as you can see in Figure 1-3. Save this file in your *Documents* folder as *sample-code.R*.

Now you can enter R code in your script file. For example, try entering `2 + 2` in the script file panel. To run a script file, press the **Run** button or use the keyboard shortcut **CMD + ENTER** on macOS and **CTRL + ENTER** on Windows. The result (4, in this case) should show up in the console pane.

You now have a working programming environment. But if you're trying to learn R, you probably want to perform more complex operations than `2 + 2`. Let's discuss 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, totals, and so on. 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. In R, your data typically comes from some external file. To work with this data in 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] "/"
>
```

Name	Size	Modified
.gitignore	51 B	Mar 21, 3:25
.Renviron	399 B	Apr 14, 11:32
Applications		
Desktop		
Documents		
Downloads		
Dropbox		
Library		
Movies		
Music		

Figure 3: RStudio with four panels

you have to run code to import it.

Importing Data

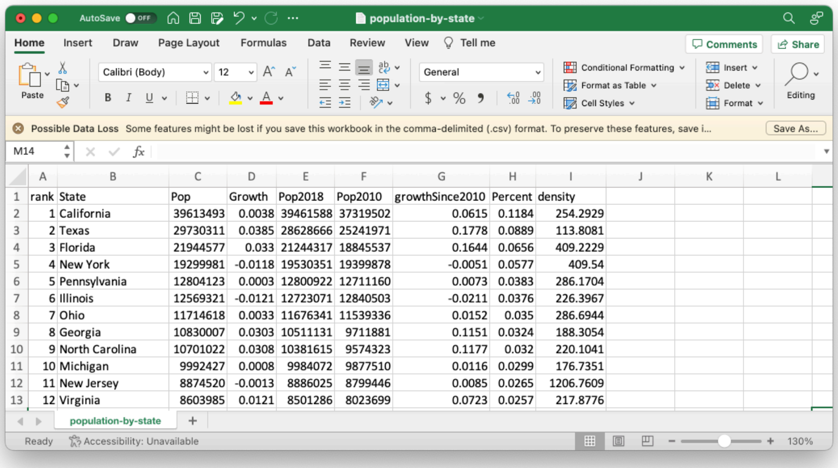
Let's import data from a *comma-separated values (CSV)* file. CSV files, a common way to store data, are text files that have values separated by commas. You can open them using most spreadsheet applications. Figure **??** shows the *population-by-state.csv* file when opened in Excel. You can download this file at https://data.rwithoutstatistics.com/population-by-state.csv. Let's import it into R.

To import the *population-by-state.csv* file into R, add a line like this one in the *sample-code.R* file, replacing the filepath with the path to the file's location on your system:

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

This line uses the `read.csv()` function. *Functions* are pieces of code that do specific things. They have a name and *arguments*, which are values that affect the function's behavior. For example, the `read.csv()` function's name is `read.csv`. Within the parentheses is the argument `file`, which specifies the file from which to import data. The text after the equal sign (`=`) gives the location of that file.

Arguments have the following structure: the argument name, followed by the equal sign and some value. Functions can have multiple arguments separated by


population-by-state
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...
Save As...
M14
Ready
Accessibility: Unavailable
130%

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

commas. For example, this code uses the `file` and `skip` arguments to import the same file but skip the first row:

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

At this point, you can run the code to import your data (without the `skip` argument). Select the line you want to run and press **Run**. The following output should show up 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
#> 11   11     New Jersey  8874520 -0.0013  8886025
#> 12   12       Virginia  8603985  0.0121  8501286
#> 13   13     Washington  7796941  0.0363  7523869
#> 14   14        Arizona  7520103  0.0506  7158024
#> 15   15      Tennessee  6944260  0.0255  6771631
#> 16   16        Massachusetts  6912239  0.0043  6882635
#> 17   17              Indiana  6805663  0.0165  6695497
#> 18   18             Missouri  6169038  0.0077  6121623
#> 19   19             Maryland  6065436  0.0049  6035802
#> 20   20             Colorado  5893634  0.0356  5691287
#> 21   21            Wisconsin  5852490  0.0078  5807406
#> 22   22            Minnesota  5706398  0.0179  5606249
#> 23   23       South Carolina  5277830  0.0381  5084156
#> 24   24              Alabama  4934193  0.0095  4887681
#> 25   25            Louisiana  4627002 -0.0070  4659690
#> 26   26             Kentucky  4480713  0.0044  4461153
#> 27   27               Oregon  4289439  0.0257  4181886
#> 28   28             Oklahoma  3990443  0.0127  3940235
#> 29   29          Connecticut  3552821 -0.0052  3571520
#> 30   30                 Utah  3310774  0.0499  3153550
#> 31   31          Puerto Rico  3194374  0.0003  3193354
#> 32   32               Nevada  3185786  0.0523  3027341
#> 33   33                 Iowa  3167974  0.0061  3148618
#> 34   34             Arkansas  3033946  0.0080  3009733
#> 35   35          Mississippi  2966407 -0.0049  2981020
#> 36   36               Kansas  2917224  0.0020  2911359
#> 37   37           New Mexico  2105005  0.0059  2092741
#> 38   38             Nebraska  1951996  0.0137  1925614
#> 39   39                Idaho  1860123  0.0626  1750536
#> 40   40        West Virginia  1767859 -0.0202  1804291
#> 41   41               Hawaii  1406430 -0.0100  1420593
#> 42   42        New Hampshire  1372203  0.0138  1353465
#> 43   43                Maine  1354522  0.0115  1339057
#> 44   44              Montana  1085004  0.0229  1060665
#> 45   45         Rhode Island  1061509  0.0030  1058287
#> 46   46             Delaware   990334  0.0257   965479
#> 47   47         South Dakota   896581  0.0204   878698
#> 48   48         North Dakota   770026  0.0158   758080
#> 49   49               Alaska   724357 -0.0147   735139
#> 50   50 District of Columbia   714153  0.0180   701547
#> 51   51              Vermont   623251 -0.0018   624358
#> 52   52              Wyoming   581075  0.0060   577601
#>     Pop2010 growthSince2010 Percent    density
#> 1  37319502          0.0615  0.1184   254.2929
#> 2  25241971          0.1778  0.0889   113.8081
#> 3  18845537          0.1644  0.0656   409.2229
#> 4  19399878         -0.0051  0.0577   409.5400
#> 5  12711160          0.0073  0.0383   286.1704
#> 6  12840503         -0.0211  0.0376   226.3967
#> 7  11539336          0.0152  0.0350   286.6944
#> 8   9711881          0.1151  0.0324   188.3054
#> 9   9574323           0.1177  0.0320   220.1041
#> 10  9877510           0.0116  0.0299   176.7351
#> 11  8799446           0.0085  0.0265  1206.7609
#> 12  8023699           0.0723  0.0257   217.8776
#> 13  6742830           0.1563  0.0233   117.3249
#> 14  6407172           0.1737  0.0225    66.2016
#> 15  6355311           0.0927  0.0208   168.4069
#> 16  6566307           0.0527  0.0207   886.1845
#> 17  6490432           0.0486  0.0203   189.9644
#> 18  5995974           0.0289  0.0184    89.7419
#> 19  5788645           0.0478  0.0181   624.8518
#> 20  5047349           0.1677  0.0176    56.8653
#> 21  5690475           0.0285  0.0175   108.0633
#> 22  5310828           0.0745  0.0171    71.6641
#> 23  4635649           0.1385  0.0158   175.5707
#> 24  4785437           0.0311  0.0147    97.4271
#> 25  4544532           0.0181  0.0138   107.0966
#> 26  4348181           0.0305  0.0134   113.4760
#> 27  3837491           0.1178  0.0128    44.6872
#> 28  3759944           0.0613  0.0119    58.1740
#> 29  3579114          -0.0073  0.0106   733.7507
#> 30  2775332           0.1929  0.0099    40.2918
#> 31  3721525          -0.1416  0.0095   923.4964
#> 32  2702405           0.1789  0.0095    29.0195
#> 33  3050745           0.0384  0.0095    56.7158
#> 34  2921964           0.0383  0.0091    58.3059
#> 35  2970548          -0.0014  0.0089    63.2186
#> 36  2858190           0.0207  0.0087    35.6808
#> 37  2064552           0.0196  0.0063    17.3540
#> 38  1829542           0.0669  0.0058    25.4087
#> 39  1570746           0.1842  0.0056    22.5079
#> 40  1854239          -0.0466  0.0053    73.5443
#> 41  1363963           0.0311  0.0042   218.9678
#> 42  1316762           0.0421  0.0041   153.2674
#> 43  1327629           0.0203  0.0040    43.9167
#> 44   990697           0.0952  0.0032     7.4547
#> 45  1053959           0.0072  0.0032  1026.6044
#> 46   899593           0.1009  0.0030   508.1242
#> 47   816166           0.0985  0.0027    11.8265
#> 48   674715           0.1413  0.0023    11.1596
#> 49   713910           0.0146  0.0022     1.2694
#> 50   605226           0.1800  0.0021 11707.4262
#> 51   625879          -0.0042  0.0019    67.6197
#> 52   564487           0.0294  0.0017     5.9847
```

This is R's way of confirming that it imported the CSV file and understands the data within it. You can see four variables, which show the rank (in terms of population size), the state name, the population, the population growth between the `Pop` and `Pop2018` variables (expressed as a percentage), and the 2018 population. There are also several other variables that are hidden in the output, though you'll see them if you import this CSV file yourself. We discuss variables in more detail in the next section.

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

Saving Data as Objects

To save your data for reuse, you need to create an object. In his book *Extending R*, John Chambers writes that "everything exists in R is an object." For our purposes, an *object* is a data structure that we store to use later. To create an object, add to your data-importing syntax so it looks like this:

```
population_data <- read.csv(file = "/Users/davidkeyes/Documents/population-by-state.cs
```

The second half of this code is the same as the line shown in the previous section, except it contains this: `<-`. Known as the *assignment operator*, it 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 and assigns it to an object called `population_data`.

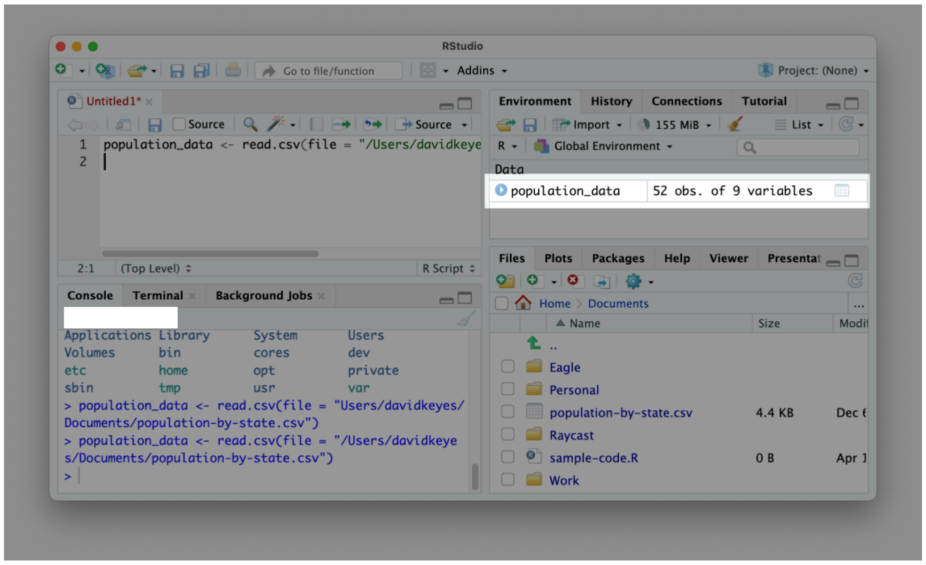
If you run this code, you should see `population_data` in your environment pane, as in Figure **??**.

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` 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 nine variables. *Variables* are the columns in a data frame, 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 we've been using comes from what is known as *base R*. This is a set of functions that are built into R, and to use them, you can simply enter their function names. However, one of the benefits of R being an



```
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")
>
```

Data	
population_data	52 obs. of 9 variables

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

Figure 5: An object in our environment pane

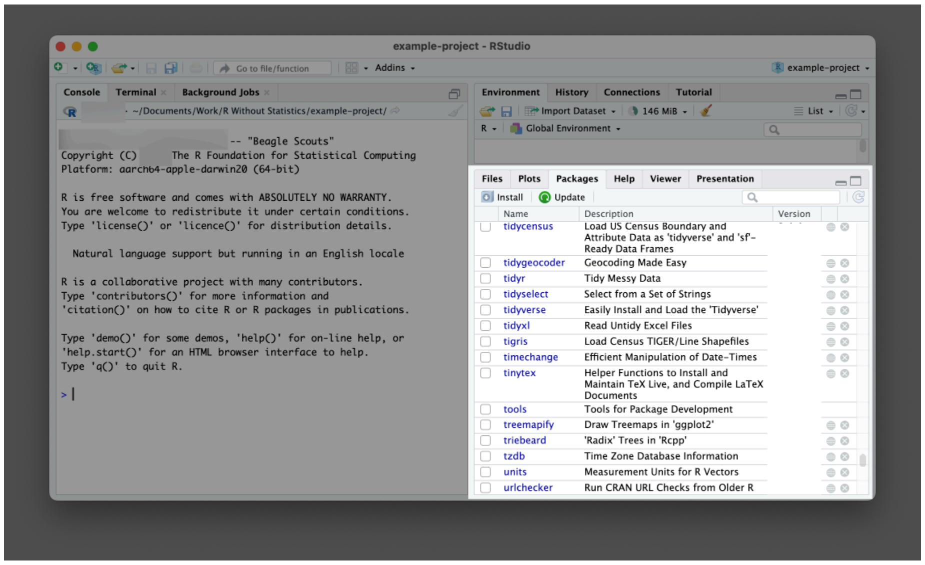
open source language is that anyone create their own code and share it with others. R users around the world make what are called *packages*, which provide their own functions to do specific things.

The best analogy for understanding packages also comes from *Modern Dive*. The functionality in base R is like the features built into a phone. A phone can do a lot on its own. But you usually want to install additional apps to do specific things. Packages are like apps, giving you specific functionality that doesn't come built into base R. In Chapters **??** and **??**, you'll create your own R package.

You can install packages using the `install.packages()` function. For example, to install the `tidyverse` package, which provides a range of functions for data import, cleaning, analysis, visualization, and more, 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 panel in R Studio. Search for tidyverse, and you should see it pop up, as in Figure **??**.

Now that you've installed `tidyverse`, let's use it. While you need to install packages only once per computer, you need to *load* packages each time you restart RStudio by running `library(tidyverse)`. Return to the *sample-code.R* file and re-import your data using a function from the tidyverse package:



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

[redacted] -- "Beagle Scouts"
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.

> |
```

Name	Description	Version
tidycensus	Load US Census Boundary and Attribute Data as 'tidyverse' and 'sf'-Ready Data Frames	
tidygeocoder	Geocoding Made Easy	
tidyr	Tidy Messy Data	
tidyselect	Select from a Set of Strings	
tidyverse	Easily Install and Load the 'Tidyverse'	
tidyxl	Read Untidy Excel Files	
tigris	Load Census TIGER/Line Shapefiles	
timechange	Efficient Manipulation of Date-Times	
tinytex	Helper Functions to Install and Maintain TeX Live, and Compile LaTeX Documents	
tools	Tools for Package Development	
treemapify	Draw Treemaps in 'ggplot2'	
triebeard	'Radix' Trees in 'Rcpp'	
tzdb	Time Zone Database Information	
units	Measurement Units for R Vectors	
urlchecker	Run CRAN URL Checks from Older R	

Figure 6: Confirmation that the tidyverse package is installed on my computer

```r
library(tidyverse)

population_data_2 <- read_csv(file = "/Users/davidkeyes/Documents/population-by-state.
```

At the top of the script, load the `tidyverse`. Then, use the package's `read_csv()` function to import the data. Note the underscore (`_`) in place of the period (`.`) in the function's name; this is a different function from one we used earlier. Using this alternate function to import CSV files achieves the same goal of creating an object, in this case one called `population_data_2`. If you enter `population_data_2` in the console, you should see this output:

```
#> # A tibble: 52 x 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  0.033  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
#> # i 42 more rows
#> # i 3 more variables: growthSince2010 <dbl>, Percent <dbl>,
#> #   density <dbl>
```

This data looks slightly different from the data we generated using the `read.csv()` function. For example, R shows us 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 are used to describe *rectangular* data like that you would see in a spreadsheet. There are some small differences between data frames and tibbles, the most important of which is that tibbles will print only the first 10 rows by default, while data frames print all rows. For the purposes of this book, we can use the terms interchangeably.

RStudio Projects

So far, we've imported a CSV file from the *Documents* folder. But the path to the file on my computer was */Users/davidkeyes/Documents/population-by-state.csv*. Because others won't have this exact location on their computer, my code won't work if they try to run it. There is a solution to this problem called *RStudio projects*.

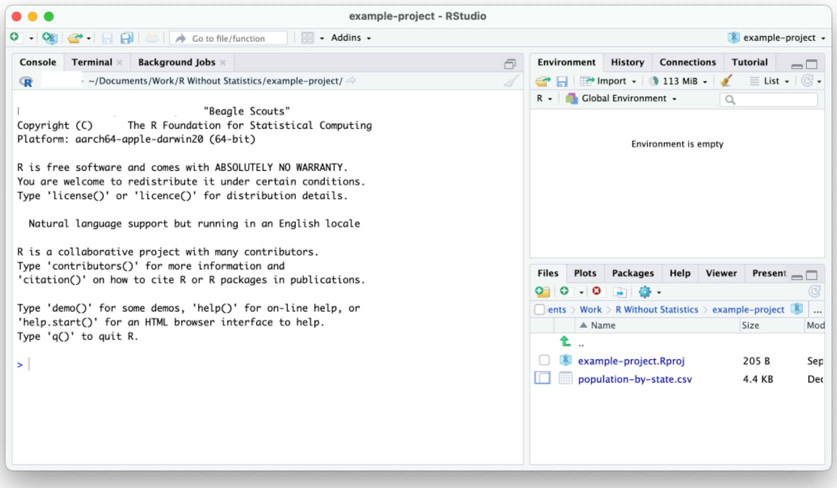
By working in a project, you can use what are known as *relative paths* to your files instead of having to write the entire filepath when calling a function to import data. If you place the CSV file in your project, anyone can open it by using the file's name, as in `read_csv(file = "population-by-state.csv")`. 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. Do this, then 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. These are for more advanced purposes.

Having created this project, you should now 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 of the example-project project. This label had previously read `Project: (None)`. If you want to make sure you're working in a project, check for its name here. Figure **??** shows these changes.

Now that you've created a project, use your operating system's filesystem to manually copy the *population-by-state.csv* file into the *example-project* directory. Once you've done this, 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. After that, remove the reference to



```
                           "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 7: RStudio with an active project

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")
```

You're able to import the *population-by-state.csv* file in this way because the RStudio project sets the working directory to be the root of your project. With the working directory set in this way, 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 we've imported data, let's do a bit of analysis on it. While I've been referring to the `tidyverse` as a single package, it is actually a collection of packages for performing data importing, analysis, visualization, and more. We'll explore several of its functions throughout this book, but this section introduces you to its basic workflow.

Tidyverse Functions

Because we've loaded the `tidyverse` package, we can access its functions. The following code calculates the mean population of all states using the `summarize()` function from the `tidyverse`:

```
summarize(.data = population_data_2,
          mean_population = mean(Pop))
```

To understand what is happening here, you need to understand two functions: `mean()` and `summarize()`. The `mean()` function calculates the mean of a set of values. If I were to write `mean(c(1, 3, 5))`, R would return `3` because that is the mean of the values `1`, `3`, and `5`. The `c()` function that surrounds the values tells R to combine these values when calculating the mean.

The `summarize()` function takes a data frame or tibble and calculates a summary of one or more variables. In the previous code, we use the `summarize()` function to calculate the mean population of all states. To do this, we pass `population_data_2` to the `.data` argument of the `summarize()` function to tell it to use that data frame. Next, we create a new variable called `mean_population` and assign it to the output of the `mean()` function run on the `Pop` variable (one of the variables in the `population_data_2` data frame).

Running this code should return a tibble with a single variable (`mean_population`) that is of type double (meaning numeric data) and has a value of `6433422`, the mean population of all states:

```
#> # A tibble: 1 x 1
#>   mean_population
#>             <dbl>
#> 1        6433422.
```

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 what's known as the `pipe` for multi-step operations. The `tidyverse` pipe, which is written as `%>%`, allows us to break steps into multiple lines. For example, we could rewrite our code using the pipe:

```
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."

The pipe becomes even more useful when we use multiple steps in our data analysis. Let's say, for example, we 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`) to include only states where the `rank` variable (which is the rank of the total population size of all states) is less than or equal to five. Then, it uses `summarize()` function, as we did before:

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

Running this code shows us the mean population of the five largest states:

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

Combining functions using the pipe lets us do multiple things to our data in a way that keeps our code readable and easy to understand.

We've introduced only two 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. *R for Data Science* by Hadley Wickham, Mine Çetinkaya-Rundel, and Garrett Grolemund is the bible of tidyverse programming and worth reading for more details on how its many packages work. Because of how useful it is, the `tidyverse` will appear in every single piece of R code you write in this book.

Comments

In addition to code, R script files often contain comments. In R script files, lines with hashes (`#`) at the start are not treated as code, but as text comments. For example, I could add a comment to the code above like so:

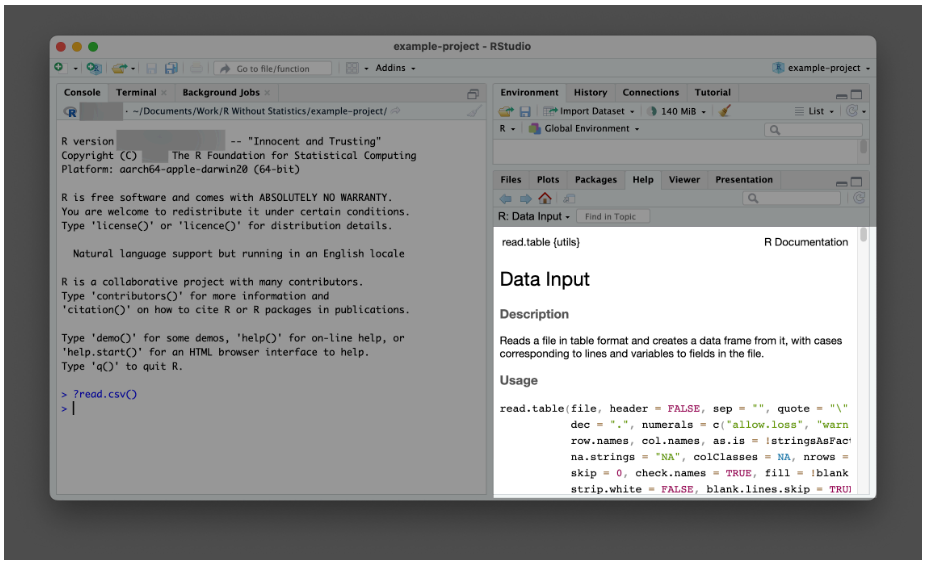
```r
# Calculate the mean population of the five largest states

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

Having this comment will help yourself and others understand what is happening in the code.

How to Get Help

Now that you've learned about 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. Learning how 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. To access the documentation for any function, simply enter `?` and then the name of the function in the console. For example, run `?read.csv` to see documentation about that function pop up in the bottom right panel, as in Figure **??**.



```
[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 = TRU
```

Figure 8: The documentation for the ‘read.csv()‘ function

Help files can be a bit hard to decipher, but at their core, they tell you what package the function comes from, what it does, what arguments it accepts, and some examples of how to use it. 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.html.

In addition to providing help files in RStudio, many R packages have documentation websites. These can be easier to read than R Studio'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.

Conclusion

This chapter should have helped you get started with R programming. You've learned a number of things, beginning with how to download and set up R and RStudio, what the various RStudio panels 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.

Lastly, we covered the basics of data exploration 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 can use R to work with your data. Let's get started!

Learn More

Consult the following resources to learn more about R programming:

Statistical Inference via Data Science: A ModernDive into R and the Tidyverse by Chester Ismay and Albert Y. Kim (CRC Press, 2020), https://moderndive.com/

The *Getting Started with R* course: https://rfortherestofus.com/courses/getting-started/

Part 1: Visualizations

Principles of Data Visualization

In the spring of 2021, nearly all of the American West was in a drought. By April of that year, officials in Southern California had declared a water emergency, citing unprecedented conditions.

This wouldn't have come as news to those living in California and other Western states. Drought conditions like those in the West in 2021 are becoming increasingly common. Yet communicating the extent of problem remains difficult. How can we show the data in a way that accurately represents it while making it 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. By working with the magazine *Scientific American* to create a data visualization of drought conditions over the last two decades in the United States, they turned to the `ggplot2` package to transform what could have been dry data (pardon the pun) into a visually arresting and impactful graph.

This chapter explores 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 recreating the drought graph step by step. In the process, we'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 data as Scherer and Karamanis, from the National Drought Center, in their stories. But Scherer and Karamanis visualized it in a way that it both grabs attention and communicates the scale of the phenomenon. Figure **??** shows a section of the final visualization. Covering four regions over the last two decades, the graph makes apparent increase in drought conditions, especially in California and the Southwest.

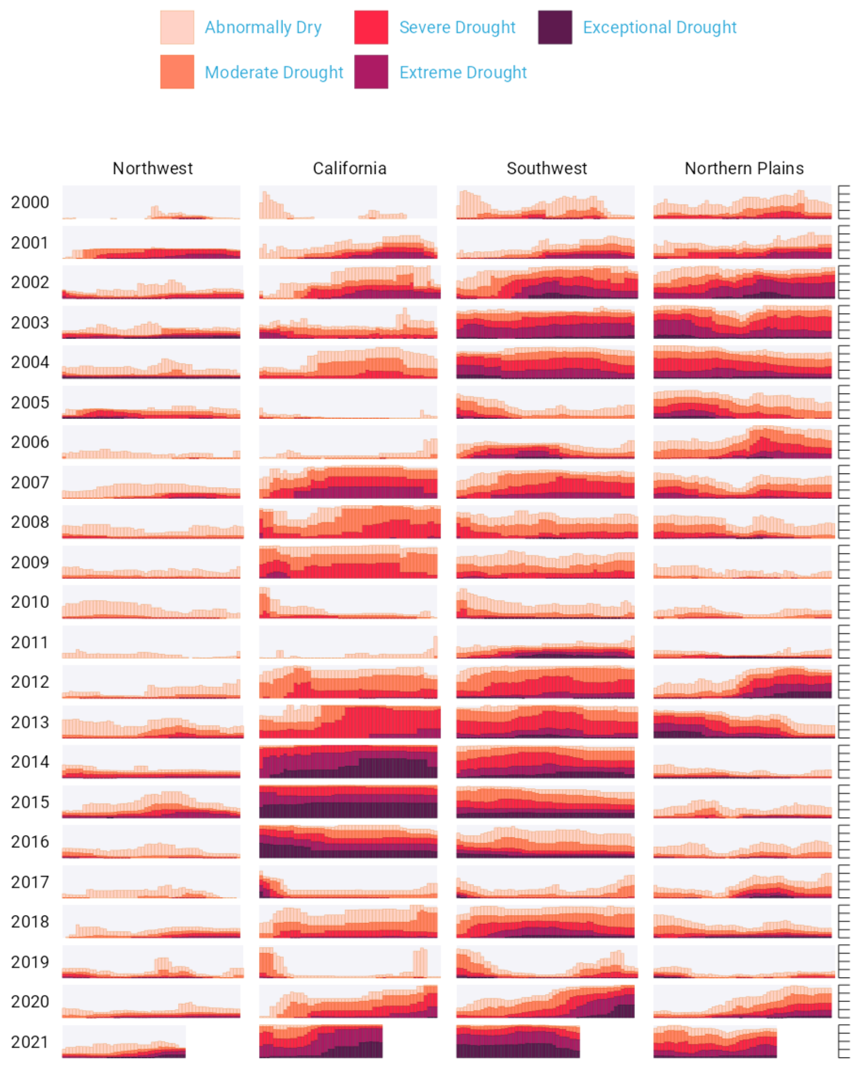


Figure 9: A section of the final drought visualization, with a few tweaks made so that the plots fit in this book

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

Need proof that Scherer and Karamanis's decluttered graph is better than the alternative? Figure **??** shows a version with a few small tweaks to the code to include grid lines and text labels on axes. Prepare yourself for clutter!

Again, it's not just that this cluttered version looks worse. The clutter actively inhibits understanding. Rather than focus on overall drought patterns (the point of the graph), our brain gets stuck reading repetitive and unnecessary axis text.

One of the best ways to reduce clutter is to break a single chart into what are known as *small multiples*. When we look closely at the data visualization, we see that it is not one chart but actually a set of charts. Each rectangle represents one region in one year. If we filter it to show the Southwest region in 2003 and add axis titles, we can see in Figure **??** that the x axis shows the week while the y axis shows the percentage of that region at different drought levels.

Zooming in on a single region in a single year also makes the color choices more obvious. The lightest bars show the percentage of the region that is abnormally dry while the darkest bars show the percentage in exceptional drought conditions. These colors, as we'll see shortly, were intentionally chosen to make differences in the drought levels visible to all readers. Even so, the R code that Scherer and Karamanis wrote to produce this complex graph is relatively simple, due largely to a theory called the grammar of graphics.

The Grammar of Graphics

If you've used Excel to make graphs, you're probably familiar with the menu shown in Figure **??**. When working in Excel, your graph-making journey begins by selecting the type of graph you want to make. Want a bar chart? Click the bar chart icon. Want a line chart? Click the line chart icon.

If you've only ever made data visualization in Excel, this first step may seem so obvious that you've never even considered the process of creating data visualization in any other way. But there are different models for thinking about graphs. Rather than conceptualizing graphs types as being distinct, we can recognize the things that they have in common and use these commonalities as the starting point for making them.

This approach to thinking about graphs comes from the late statistician Leland

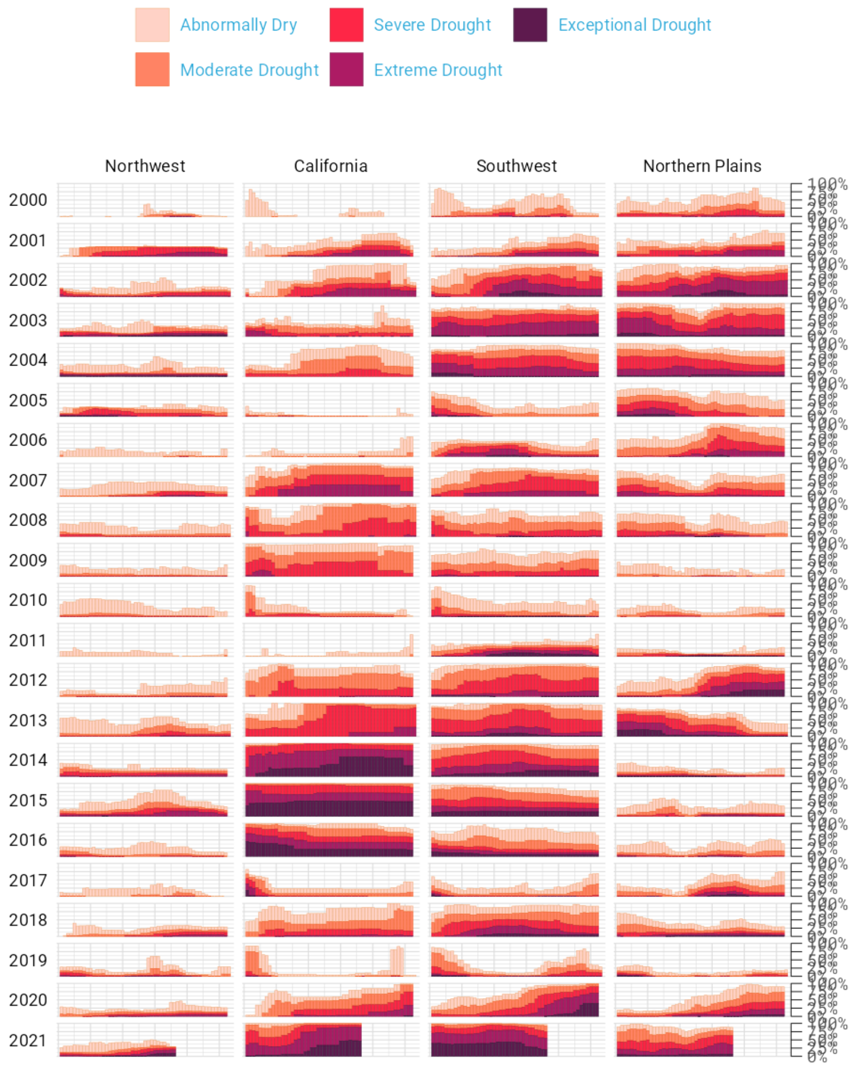


Figure 10: The cluttered version of the drought visualization

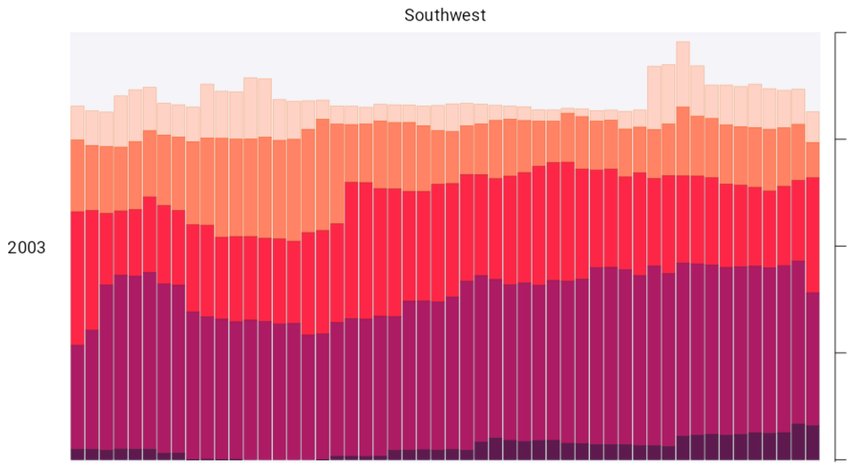


Figure 11: A drought visualization for the Southwest in 2003

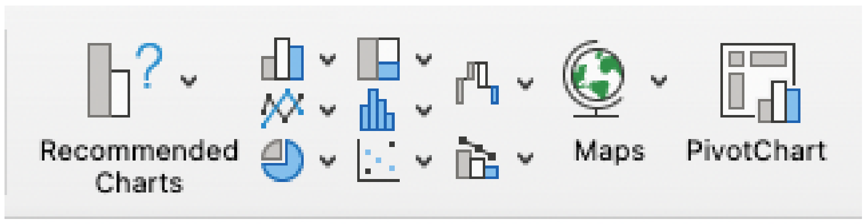


Figure 12: The Excel chart chooser menu

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* 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), knowledge of the grammar of graphics allows us to understand why certain graph types "work."

Thinking about data visualization through the lens of the grammar of graphics allow us to see, 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 no matter whether the graph is a bar chart or a line chart, for example. Consider Figure **??**, which shows two charts that use identical data on life expectancy in Afghanistan.

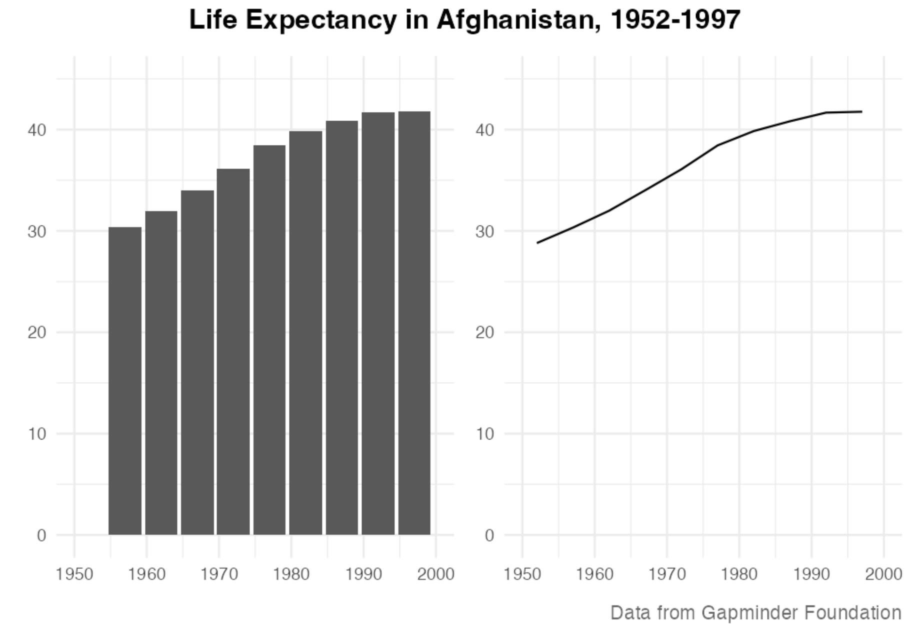


Figure 13: A bar chart and a line chart showing identical data on Afghanistan life expectancy

While they look different (and would, to the Excel user, be different types of graphs), Wilkinson's grammar of graphics allows us to see 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 an article titled "A Layered Grammar of Graphics." By providing the tools to implement Wilkinson's ideas, `ggplot2` would come to revolutionize the world of data visualization.

Working With ggplot2

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. Let's walk through some of the most important ones. We'll begin by selecting variables to map to aesthetic properties. Then we'll choose a geometric object to use to represent our data. Next, we'll change the aesthetic properties of our chart (its color scheme, for example) using a `scale_` function. Finally, we'll use a `theme_` function to set the overall look-and-feel of our plot.

The First Layer: Mapping Data to Aesthetic Properties

When creating a graph with ggplot, we begin by mapping data to aesthetic properties. All this really means is that we use things like the x or y axis, color, and size (the so-called *aesthetic properties*) to represent variables. To make this concrete, we'll use the data on life expectancy in Afghanistan, introduced in the previous section, to generate a plot. Access this data with the following code:

```r
library(tidyverse)

gapminder_10_rows <- read_csv("https://data.rwithoutstatistics.com/gapminder_10_rows.csv")
```

Here's what the `gapminder_10_rows` data frame looks like:

```
#> # A tibble: 10 x 6
#>    country     continent  year lifeExp      pop gdpPercap
#>    <chr>       <chr>     <dbl>   <dbl>    <dbl>     <dbl>
#>  1 Afghanistan Asia       1952  28.801  8425333    779.45
#>  2 Afghanistan Asia       1957  30.332  9240934    820.85
#>  3 Afghanistan Asia       1962  31.997 10267083    853.10
#>  4 Afghanistan Asia       1967  34.02  11537966    836.20
#>  5 Afghanistan Asia       1972  36.088 13079460    739.98
#>  6 Afghanistan Asia       1977  38.438 14880372    786.11
#>  7 Afghanistan Asia       1982  39.854 12881816    978.01
#>  8 Afghanistan Asia       1987  40.822 13867957    852.40
#>  9 Afghanistan Asia       1992  41.674 16317921    649.34
#> 10 Afghanistan Asia       1997  41.763 22227415    635.34
```

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

If we want to make a chart with ggplot, we need to first decide which variable to put on the x axis and which to put on the y axis. For data showing change over time, it is common to put the date on the x axis and the value of what you are showing on the y axis. That means we would use the variable `year` on the x axis and the variable `lifeExp` on the y axis. To do so, we begin by using the `ggplot()` function:

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

Within this function, we tell R that we're using the data frame `gapminder_10_rows`. We also map `year` to the x axis and `lifeExp` to the y axis.

When we run the code, what we get in Figure **??** doesn't look like much.

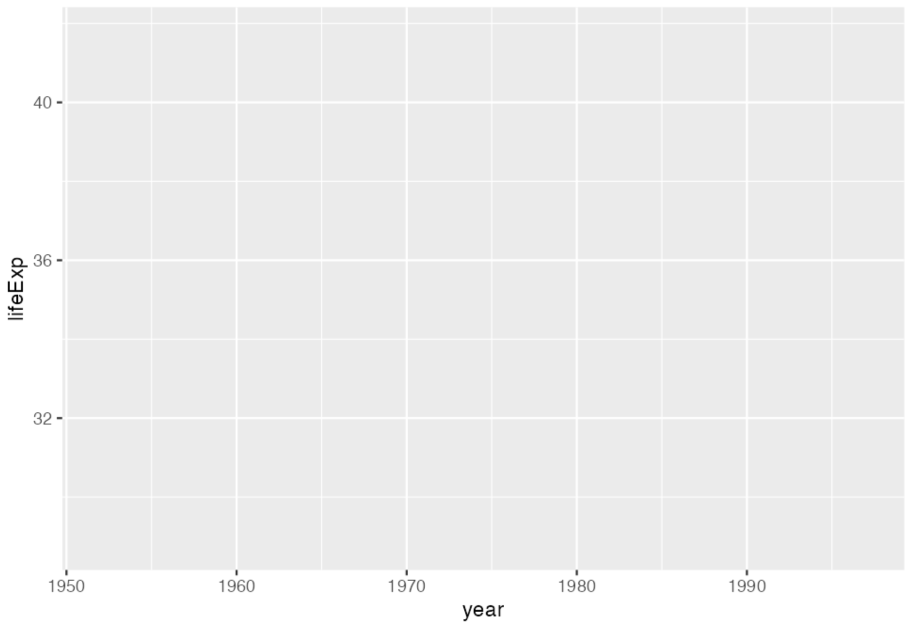


Figure 14: A blank chart

If you look closely, however, you should see that the x axis corresponds to `year` and the y axis corresponds to `lifeExp`. Also, the values on the x and y axes match the scope of our 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 seems to have been created with this data in mind (because it was). Likewise, `lifeExp`, which goes from about 28 to about 42, will fit nicely on our y axis.

The Second Layer: Choosing the geoms

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

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

Now, in Figure **??**, we see that people in 1952 had a life expectancy of about 28 and that this value rose every year included in the data.

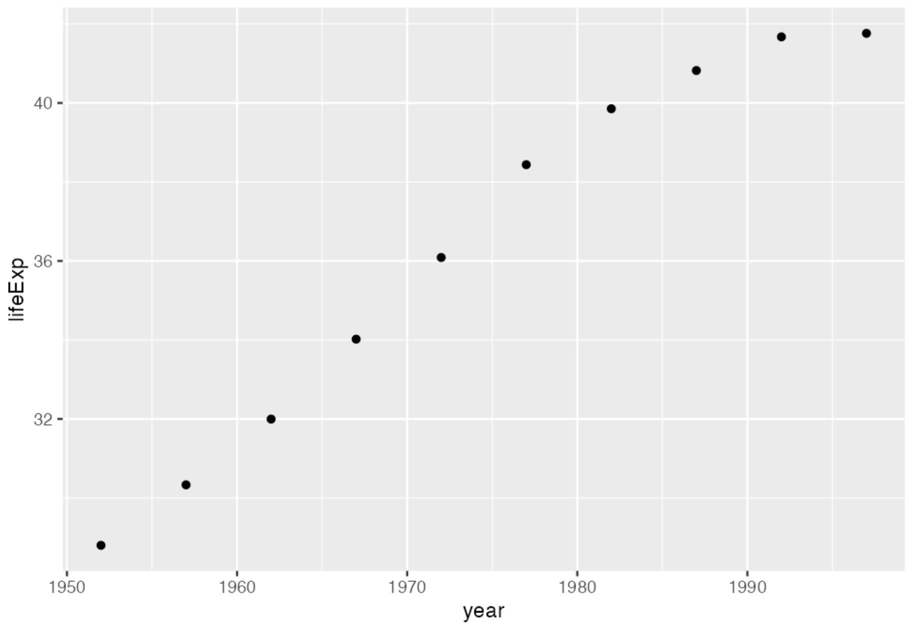


Figure 15: The same chart but with points added

Let's say we change our mind and want to make a line chart instead. All we have to do is replace `geom_point()` with `geom_line()`:

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

Figure **??** shows the result.

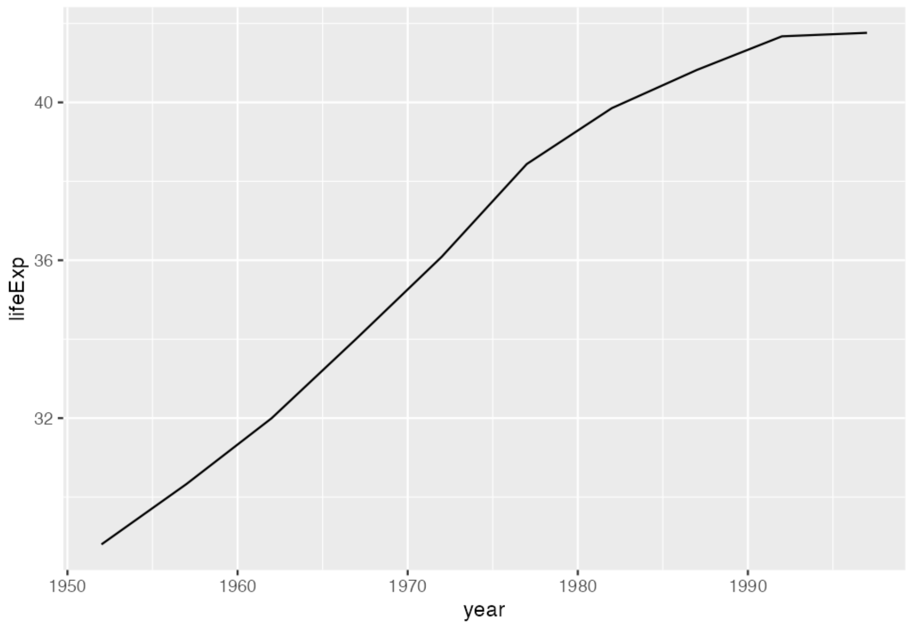


Figure 16: The data as a line chart

To really get fancy, what if we add both `geom_point()` and `geom_line()`?

```
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 **??**.

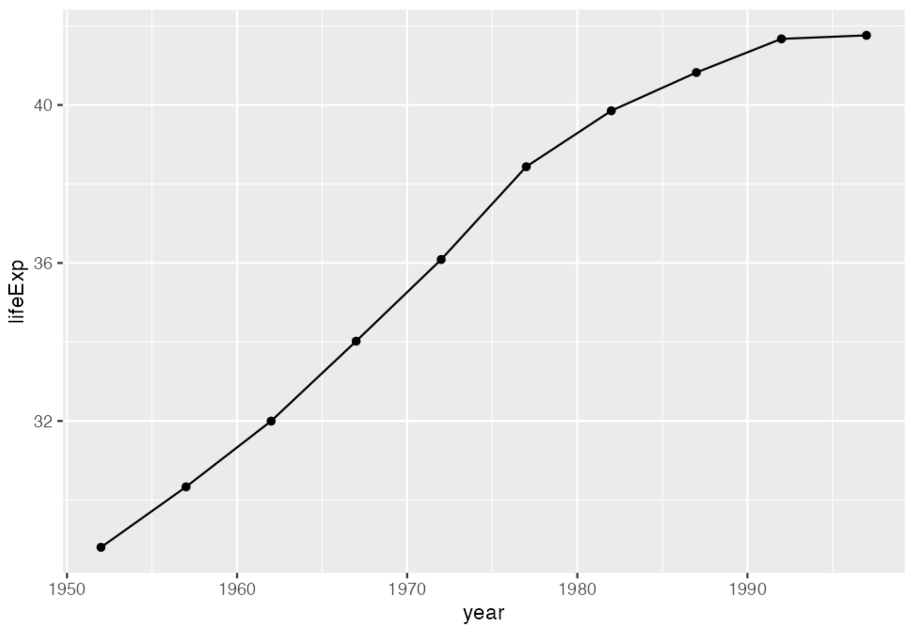


Figure 17: The data with points and a line

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

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

Note in Figure **??** that the y axis range has been automatically updated, going from 0 to 40 to account for the different geom.

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 us think. Both can have the same underlying properties (namely, putting years on the x axis and life expectancies on the y axis). They simply use different geometric objects to visually represent the data.

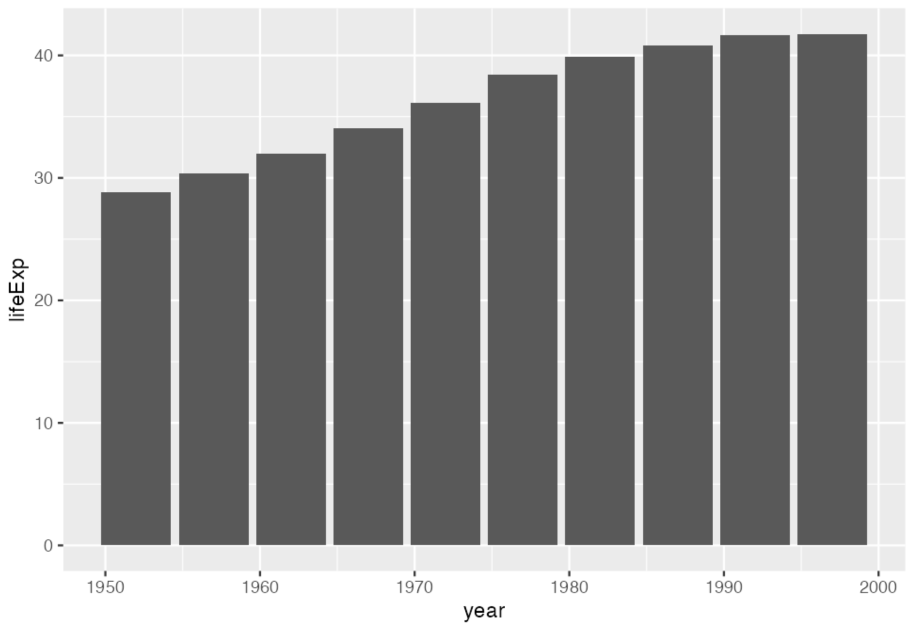


Figure 18: The data as a bar chart

The Third Layer: Altering Aesthetic Properties

Before we return to the drought data visualization, let's look at a few additional layers that can help us can alter the bar chart. Say we 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`. (Slightly confusingly, the aesthetic property of `color` would, for a bar chart, change only the outline of each bar). In the same way that we mapped `year` to the x axis and `lifeExp` to the y axis, we 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 **??** shows the result. We see now that, for earlier years, the fill is darker, while for later years, it is lighter (the legend, added to the right of our plot, also indicates this).

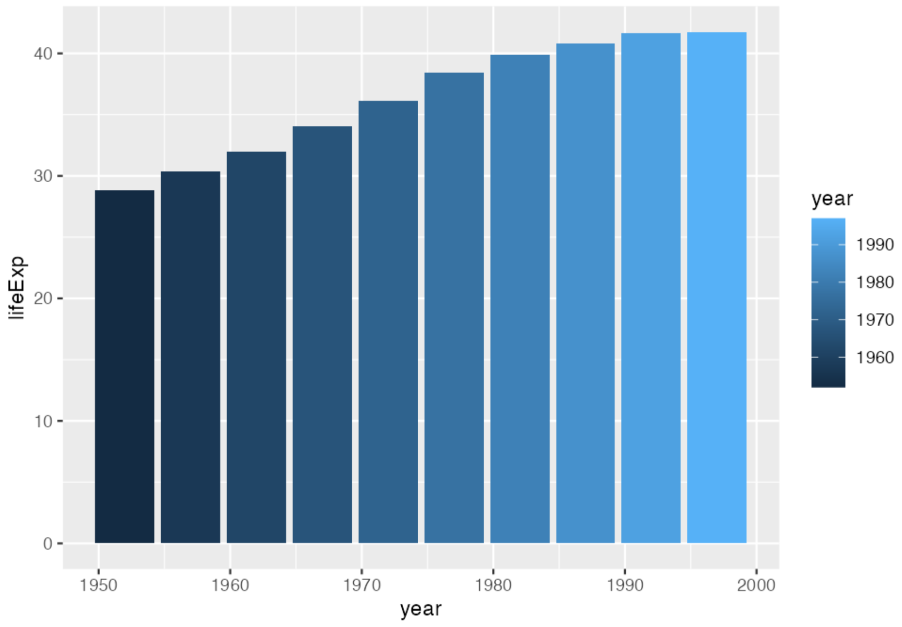


Figure 19: The same chart, now with added colors

What if we wanted to change the fill colors? For that, we use a new scale layer. To do this, I'll use 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.