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.

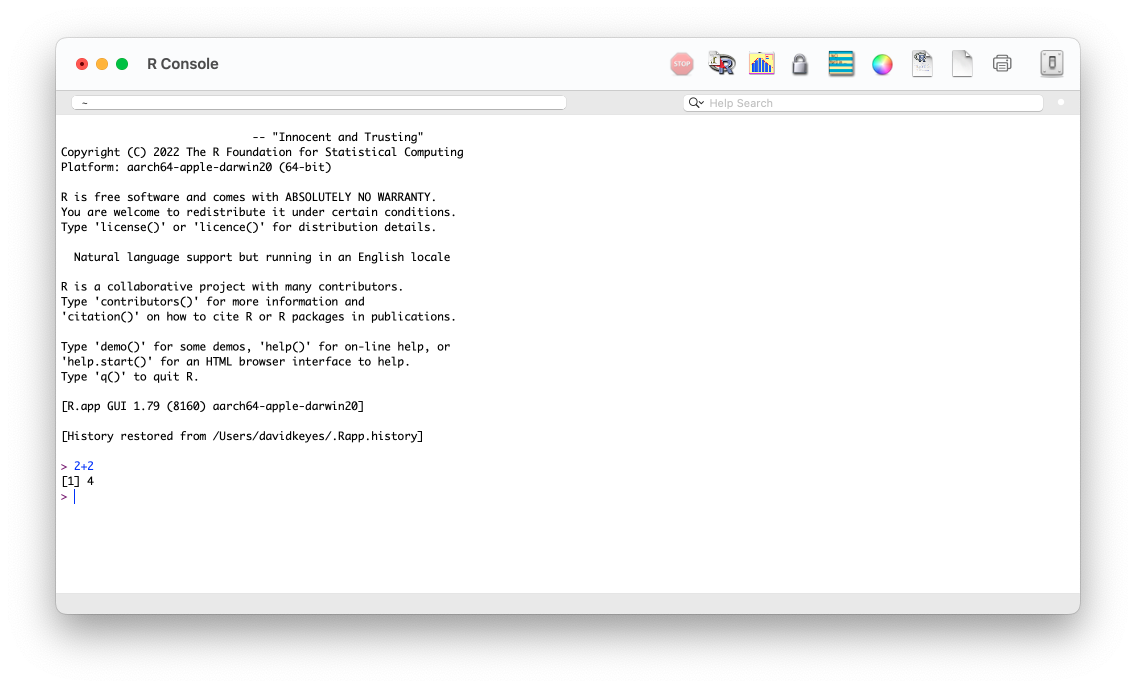
This chapter won’t provide a complete introduction to R programming; rather, it will focus on the knowledge you need to follow along with the rest of the book. If you have prior experience with R, feel free to skip this chapter, but if you’re just starting out, it should help you make sense of the rest 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 (CRC Press, 2019) 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 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.

[F01001.png]

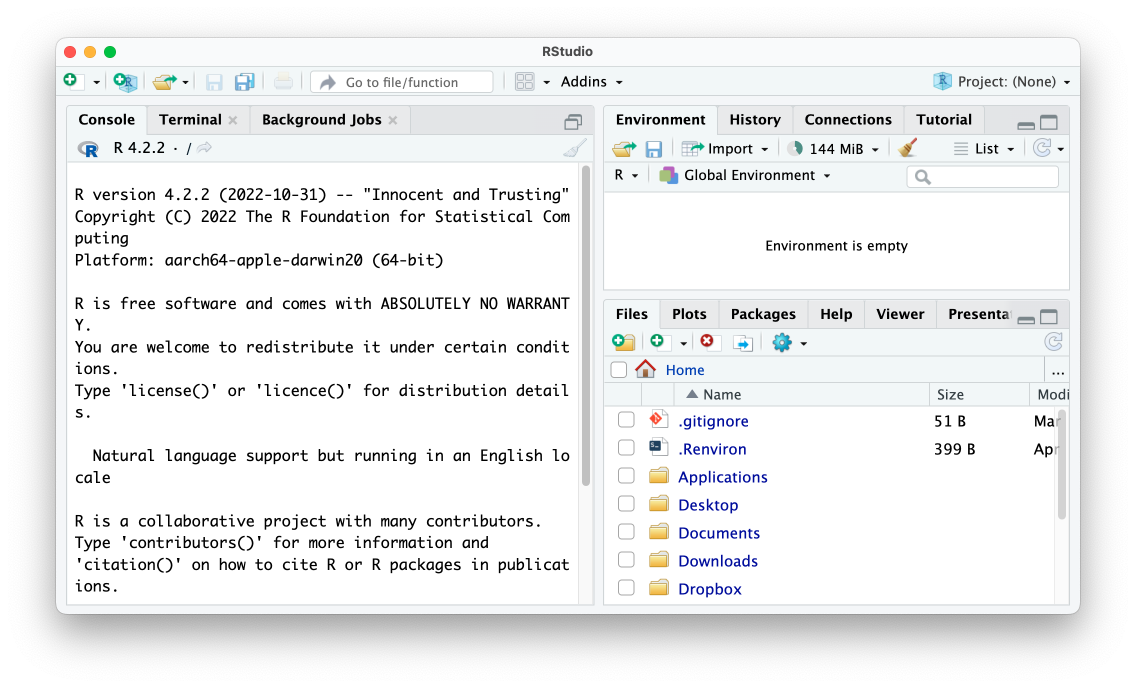
* + - * 1. The R console

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 panels shown in Figure 1-2.

[F01002.png]



* + - * 1. The RStudio editor

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 in “Objects” on page XX.

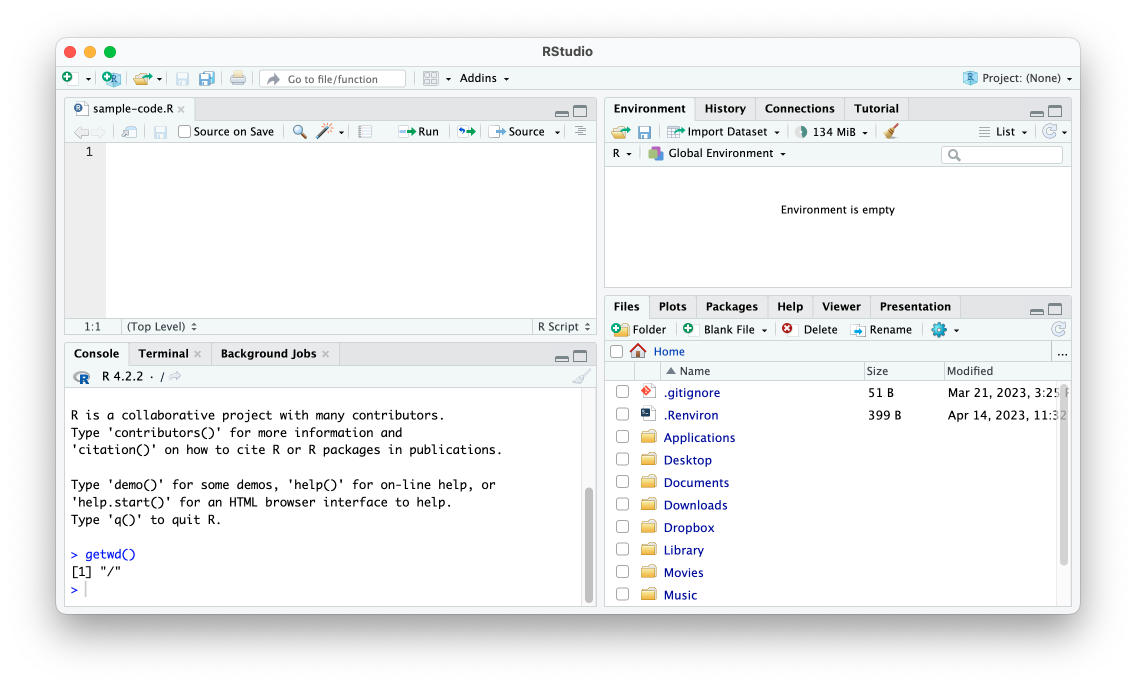
There is one more panel that you’ll typically use when working in RStudio, but to make it appear, you need to create an R script file.

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 re-create that code from scratch tomorrow. Writing your code in files lets you 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 File4New File4R Script, and a fourth panel should appear in the top left of R Studio, as shown in Figure 1-3. Save this file in your Documents folder as sample-code.R.

[F01003.png]



* + - * 1. Creating an R script (top-left panel)

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

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. Let’s 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 how to do simple calculations will prepare you to do more serious data analysis tasks later in this chapter. Let’s cover some of these basics.

Arithmetic Operators

As you just saw, R can use common arithmetic operators such as +. These include - 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 we enter. While you don't need to add spaces around operators, as we’ve done here, adding spaces makes your code much more readable.

Like when using a calculator, you can 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 adds two plus one before multiplying it by two in order to get six.

R has more advanced arithmetic operators for tasks such as calculating exponents, done with the \*\* operator as follows:

2 \* 3

8

Multiplying two to the third power returns eight. We can also get the remainder of a division operation using the %% operator:

10 %% 3

1

Diving ten by three, we get a remainder of one, which R returns for us.

You won’t need to use these advanced arithmetic operators for the activities in this book, but they’re good to know about 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 do things like keep data only where a value is greater than a certain threshold. You’ll see comparison operators used in this way in “Tidyverse Functions” on page XX.

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.

Take the print() function, which displays results. Within the print() function, we can use the x argument to give it a number to display. This code will print the number 1.1.

> print(x = 1.1)

1.1

We can also use the digits argument to determine how many digits are printed. This code, for example, will only display one digit (in other words, a whole number):

> print(x = 1.1, digits = 1)

1

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

You can find a list of all functions built into R at https://stat.ethz.ch/R-manual/R-devel/library/base/html/00Index.html.

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:

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

20

Running this code in the console returns the value 20. The name of the mean() function is mean, and it uses one argument, x, which provides the values to use in the calculation.

A function’s arguments have the following structure: the argument name, followed by the equal sign (=) and some value. 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.

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 another one nested within it, for c().

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 this function’s help file.

Next, let’s discuss how to import data for your R programs to work with.

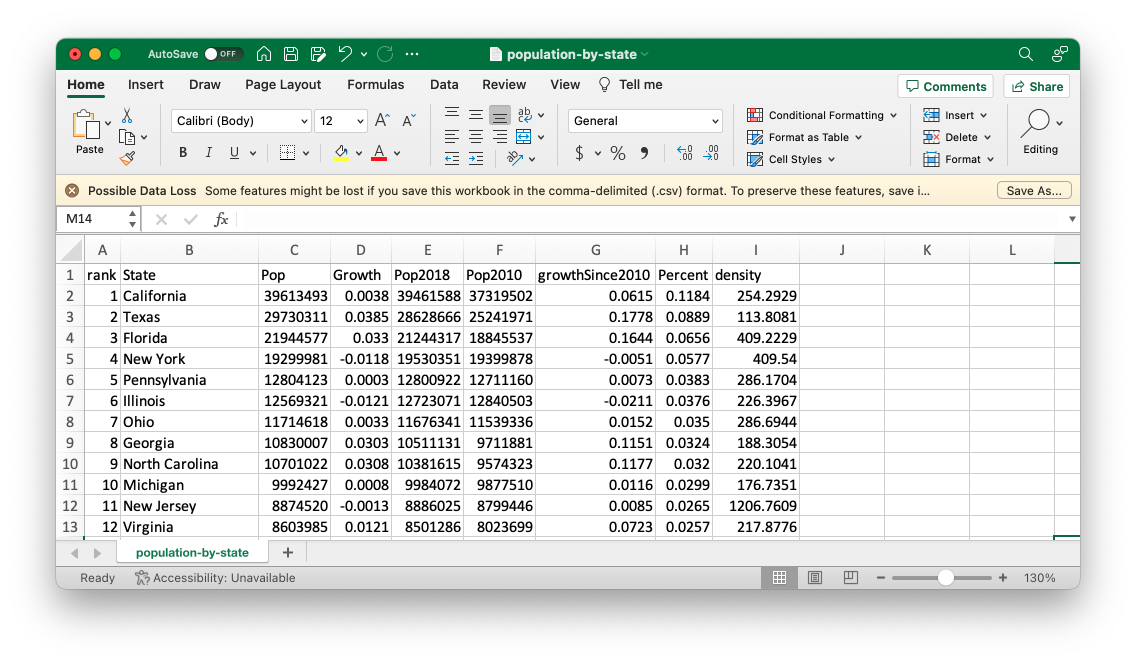
Working with Data

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. 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

Let’s import data from a comma-separated values (CSV) file. CSV files, a common way to store data, are text files that hold series of related values separated by commas. You can open them using most spreadsheet applications, which use columns rather than commas as separators. For example, Figure 1-4 shows the population-by-state.csv file when opened in Excel.

[f01004.png]



* + - * 1. The population-by-state.csv file in Excel

To instead work with this file in R, download it from <https://data.rwithoutstatistics.com/population-by-state.csv>. Save it to a location on your computer (for example, your Documents folder).

Let’s now import it into R. To do so, add a line like this one in thesample-code.R file you created earlier in this chapter, 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. Its file argument provides the path to the file to open.

This 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-state.csv](https://data.rwithoutstatistics.com/population-by-state.csv)", skip = 1)

To learn about additional arguments, enter ?read.csv() in the console to see this function’s 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 panel in RStudio and press Run. The following output should appear 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, population, population growth between the Pop and Pop2018 variables (expressed as a percentage), and the 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 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

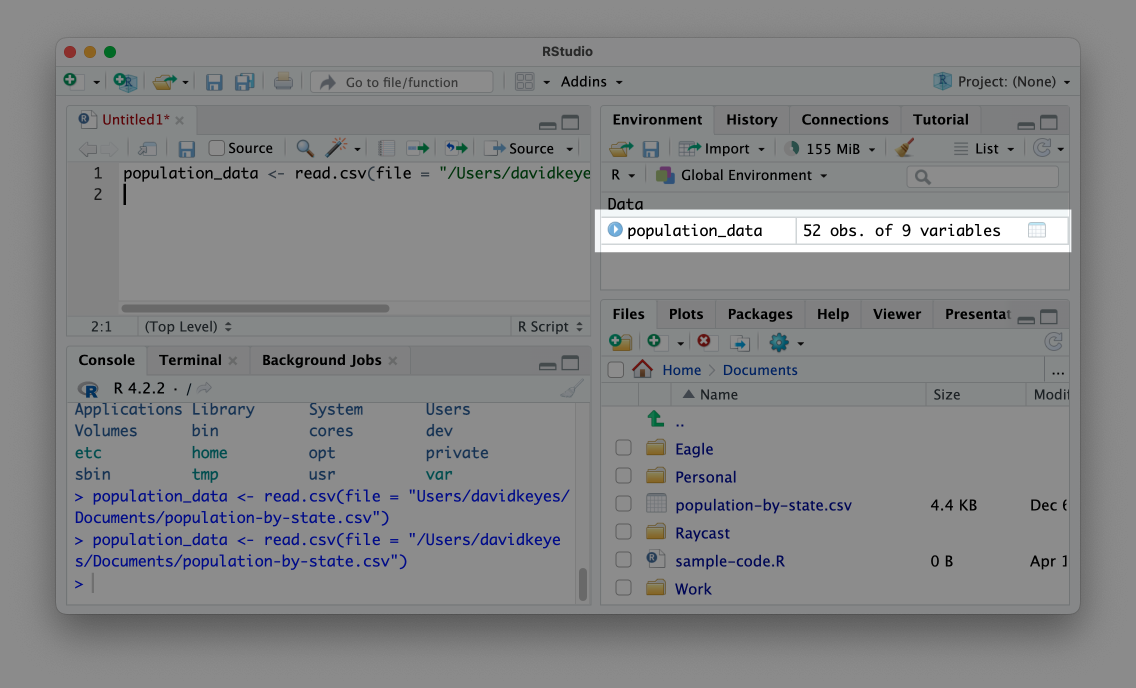
To save your data for reuse, you need to create an object. For our purposes, an object is a data structure that we store to use later. To create an object, add to your data-importing syntax so it looks like this:

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

The second half of this code is the same as the line shown in the previous section, except it 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.

If you run this code, you should see population\_data in your Environment pane, as in Figure 1-5.

[F01005.png]



* + - * 1. An 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 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 base R, which is the set of built-in R functions. To use them, simply enter the function names. The mean() and c() functions used earlier are also part of base R. However, one of the benefits of R being an 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 12, 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 thePackages tab on the bottom right panel in R Studio. Search for tidyverse, and you should see it pop up.

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:

library(tidyverse)

population\_data\_2 <- read\_csv(file = "/Users/davidkeyes/Documents/population-by-state.csv")

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 the 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 × 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 rectangulardata like what 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

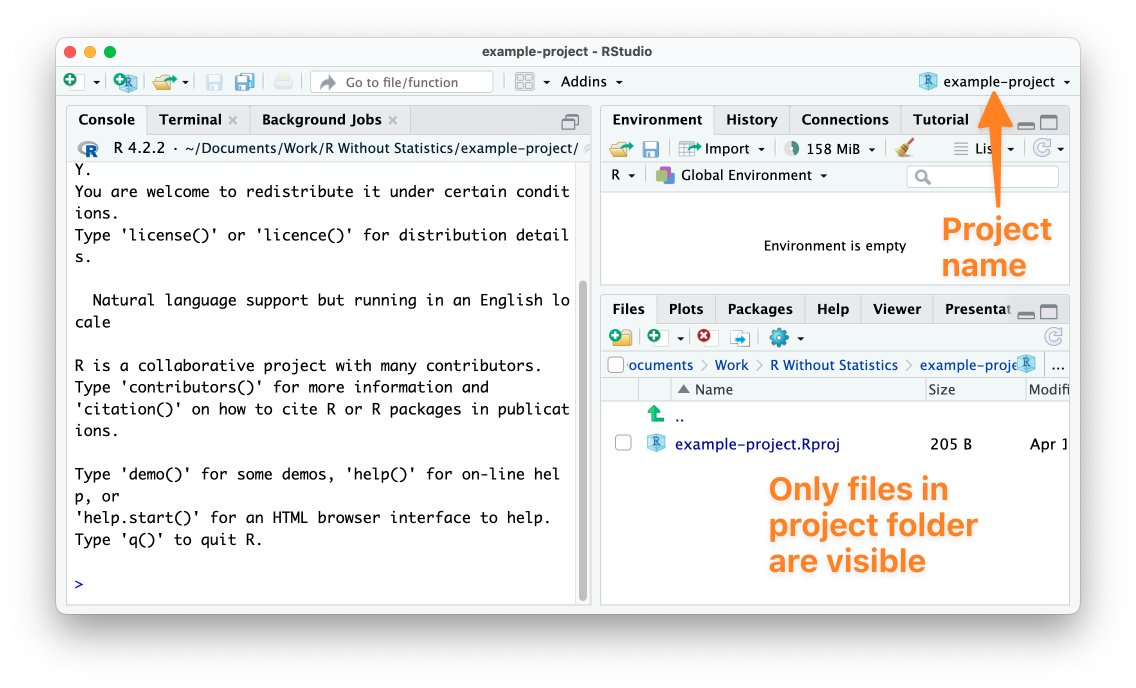
So far, we’ve imported a CSV file from the Documents folder. But the path to the file on my computer was /Users/davidkeyes/Documents/population-by-state.csv. Because others won’t have this exact location on their computer, my code won’t work if they try to run it. There is a solution to this problem 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 File4New 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 1-7 shows these changes.

[F01007.png]



* + - * 1. 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 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 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. For example, 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))

The summarize() function essentially takes a data frame or tibble and calculates some piece of information for one or more of the variables in that data set. In this code, we use the summarize() function to calculate the mean population of all states.

To do this, we first pass population\_data\_2 to the .data argument of the summarize() function to tell it to use that data frame to perform the calculation. Next, we create a new variable called mean\_population and assign it to the output of the mean() function we introduced earlier. We run mean() on the Pop variable, one of the variables in the population\_data\_2 data frame.

You might be wondering why we don’t need to use the c() function within mean(), as we did earlier in this chapter. The reason is that we’re passing the function only one argument here: Pop, which contains the set of population data for which we want to calculate the mean. So, we don’t need to use c() to combine multiple values into one.

Running this code should return a tibble with a single variable (mean\_population):

#> # A tibble: 1 × 1

#> mean\_population

#> <dbl>

#> 1 6433422.

The variable is of type double, used to hold general numeric data. Other common data types are integer (for whole numbers, such as the values 4, 82, and 915), character (for text values), and logical (for the TRUE/FALSE values we encountered when using comparison operators). 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. 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 what’s known as the pipe for multistep 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.”

Note that the line following the pipe is indented. RStudio will automatically add two spaces to the start of lines that follow pipes, which makes the code easier to read.

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 is less than or equal to (<=) five. Then, it uses summarize() function, as we did before, to calculate the mean of those states:

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 × 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. Indentation can also make our code readable.

We’ve introduced 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. R for Data Science(O’Reilly, 2023)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 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))

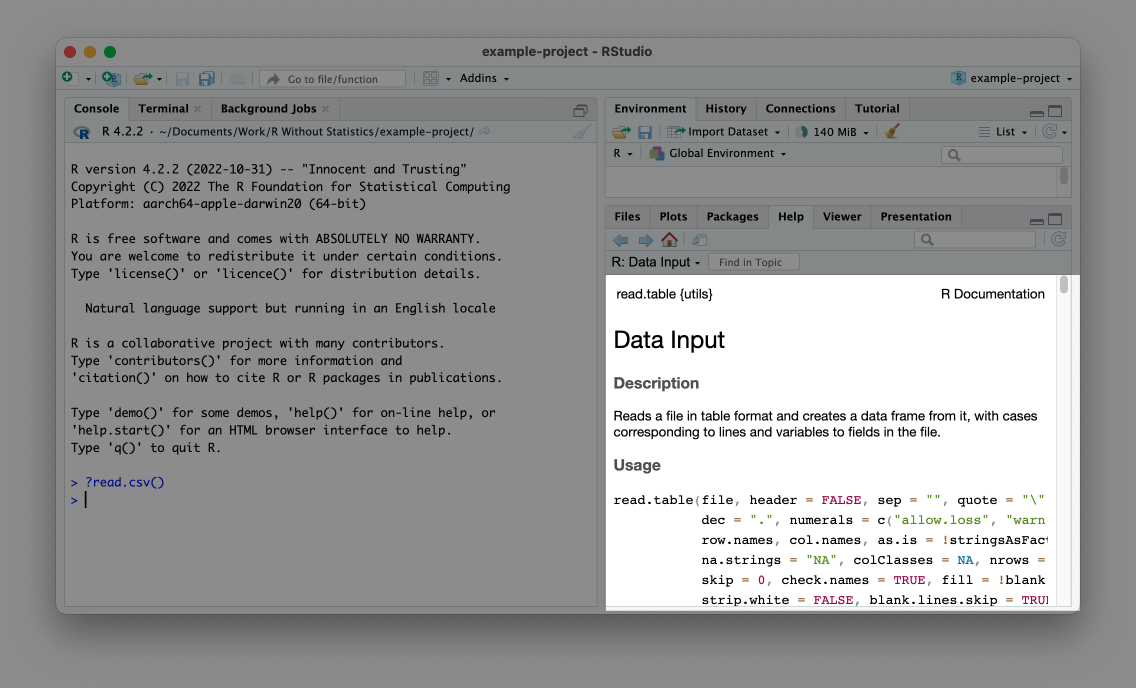
Having this comment will help yourself and others understand what is happening in the code.

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. (We’ve already done this for a couple of the functions in this chapter.) For example, run ?read.csv to see documentation about that function pop up in the bottom right panel, as in Figure 1-7.

[F01008.png]



* + - * 1. 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 (Princeton University Press, 2018). 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/