An R Programming Crash Course

R has a well-earned reputation for being hard to learn, especially for those who come to it without prior programming experience. This chapter is designed to help anyone who has never used R before. You’ll set up an R programming environment with RStudio and learn how to 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 used in this book.

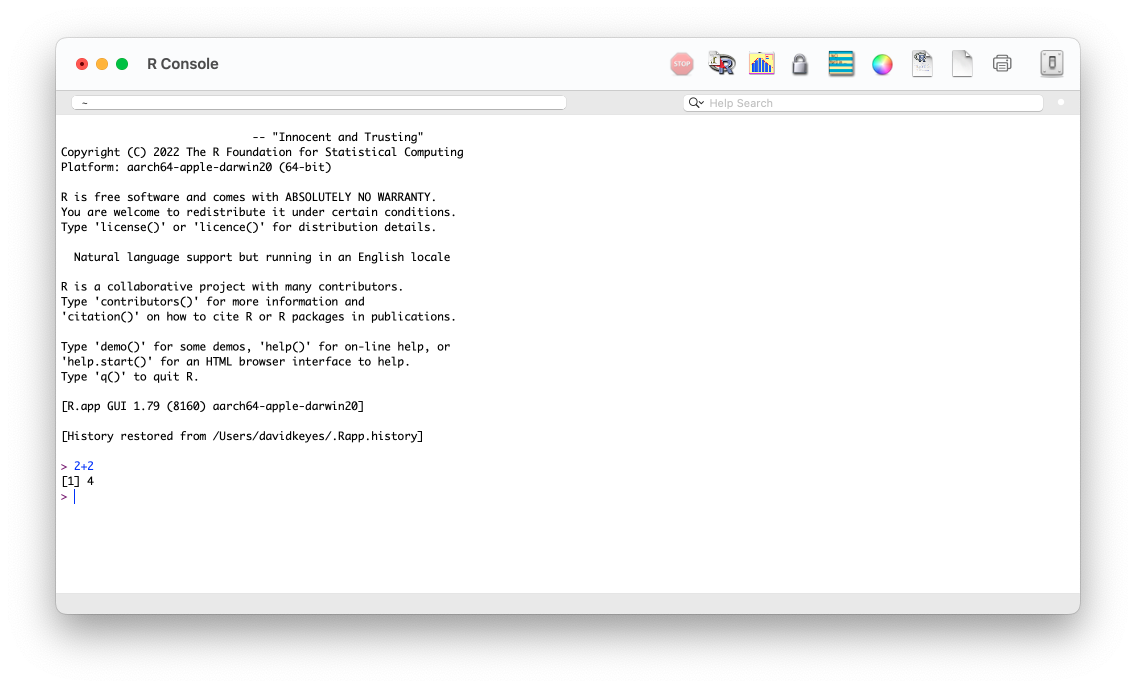
This chapter won’t provide a complete introduction to R programming; rather, it will focus on the knowledge you need to follow along with the rest of the book. If you have prior experience with R, feel free to skip ahead to Chapter 2.

Setting Up

You’ll need two pieces of software to use R effectively. The first is R itself, which provides the underlying computational tools that make the language work. The second is an integrated development environment (IDE) like RStudio. This coding platform simplifies working with R. The best way to understand the relationship between R and RStudio is with this analogy from Chester Ismay and Albert Kim’s book Statistical Inference via Data Science: A ModernDive into R and the Tidyverse (CRC Press, 2019): R is the engine that powers your data, and RStudio is like the dashboard that provides a user-friendly interface.

Installing R and RStudio

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

[F01001.png]

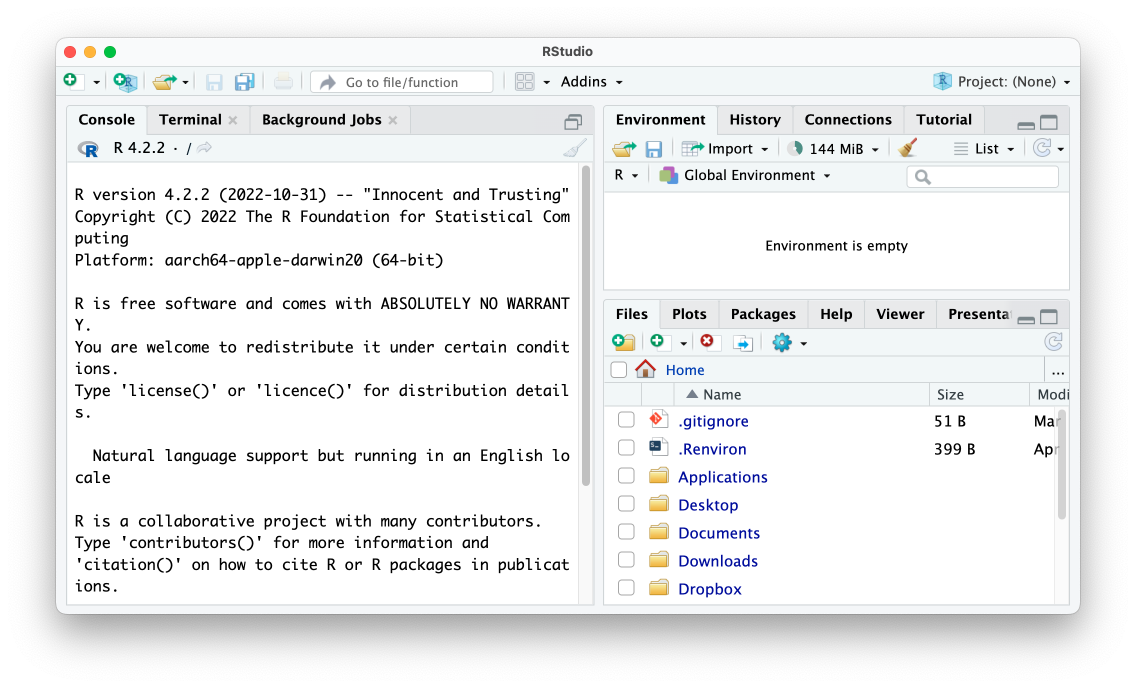
* + - * 1. The R console

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 panes shown in Figure 1-2.

[F01002.png]



* + - * 1. The RStudio editor

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

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

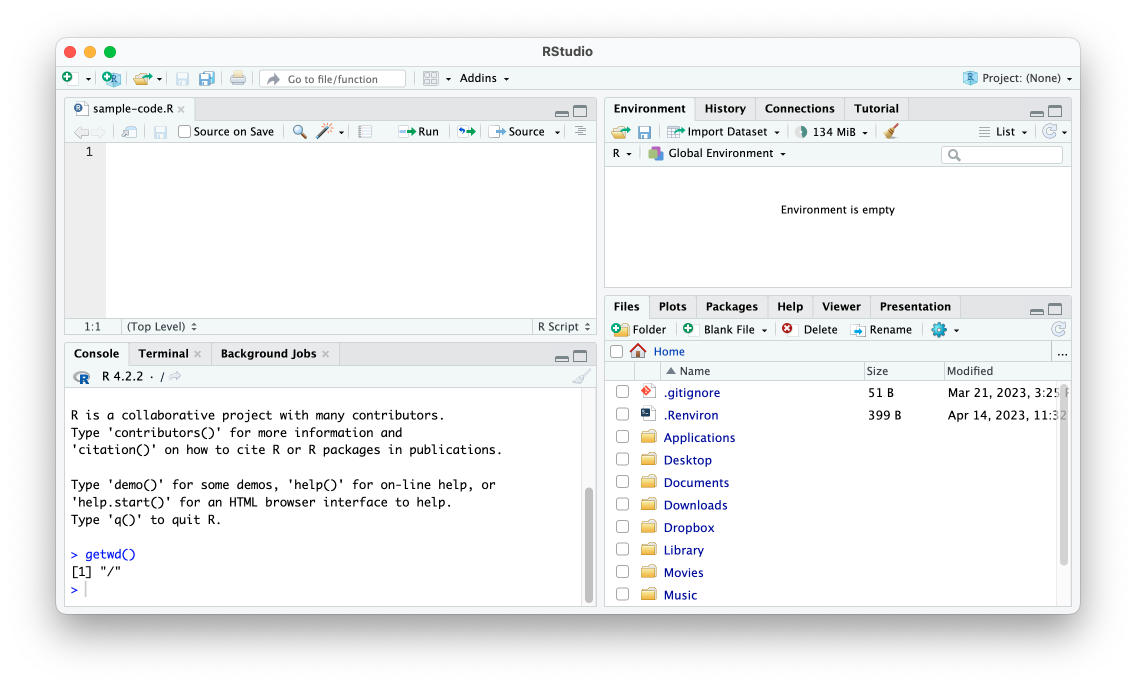
There is one more pane that you’ll typically use when working in RStudio, but to see it, first you need to create an R script file.

R Script Files

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

R script files, which use the .R extension, save your code so you can run it later. To create an R script file, go to File4New File4R Script, and the script file pane should appear in the top left of RStudio, as shown in Figure 1-3. Save this file in your Documents folder as sample-code.R.

[F01003.png]



* + - * 1. The script file pane (top left)

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

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

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

Basic R Syntax

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

Arithmetic Operators

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

> 2 - 1

1

> 3 \* 3

9

> 16 / 4

4

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

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

> 2 \* (2 + 1)

6

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

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

> 2 \*\* 3

8

This is equivalent to 23, which returns 8.

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

> 10 %% 3

1

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

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

Comparison Operators

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

> 2 > 1

TRUE

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

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

> 498 == 498

TRUE

> 2 != 2

FALSE

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

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

Consider the print() function, which displays information:

> print(x = 1.1)

1.1

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

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

> print(x = 1.1, digits = 1)

1

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

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

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

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

20

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

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

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

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

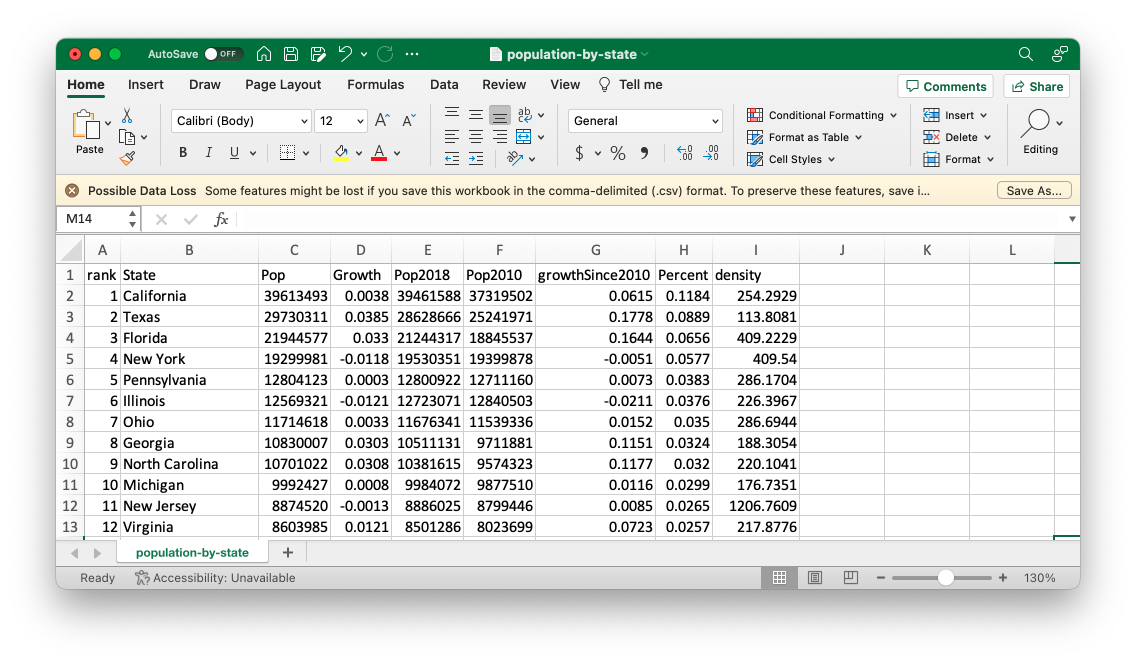
Working with Data

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

Importing Data

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

[f01004.png]



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

To 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, such as your Documents folder.

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

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

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

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

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

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

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

#> rank State Pop Growth Pop2018

#> 1 1 California 39613493 0.0038 39461588

#> 2 2 Texas 29730311 0.0385 28628666

#> 3 3 Florida 21944577 0.0330 21244317

#> 4 4 New York 19299981 -0.0118 19530351

#> 5 5 Pennsylvania 12804123 0.0003 12800922

#> 6 6 Illinois 12569321 -0.0121 12723071

#> 7 7 Ohio 11714618 0.0033 11676341

#> 8 8 Georgia 10830007 0.0303 10511131

#> 9 9 North Carolina 10701022 0.0308 10381615

#> 10 10 Michigan 9992427 0.0008 9984072

--snip--

This is R’s way of confirming that it imported the CSV file and understands the data within it. Four variables show each state’s rank (in terms of population size), name, 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 ready to work with your data now, but all you’ve really done at this point is display the result of running the code that imports the data. To actually use the data, you need to save it to an object.

Saving Data as Objects

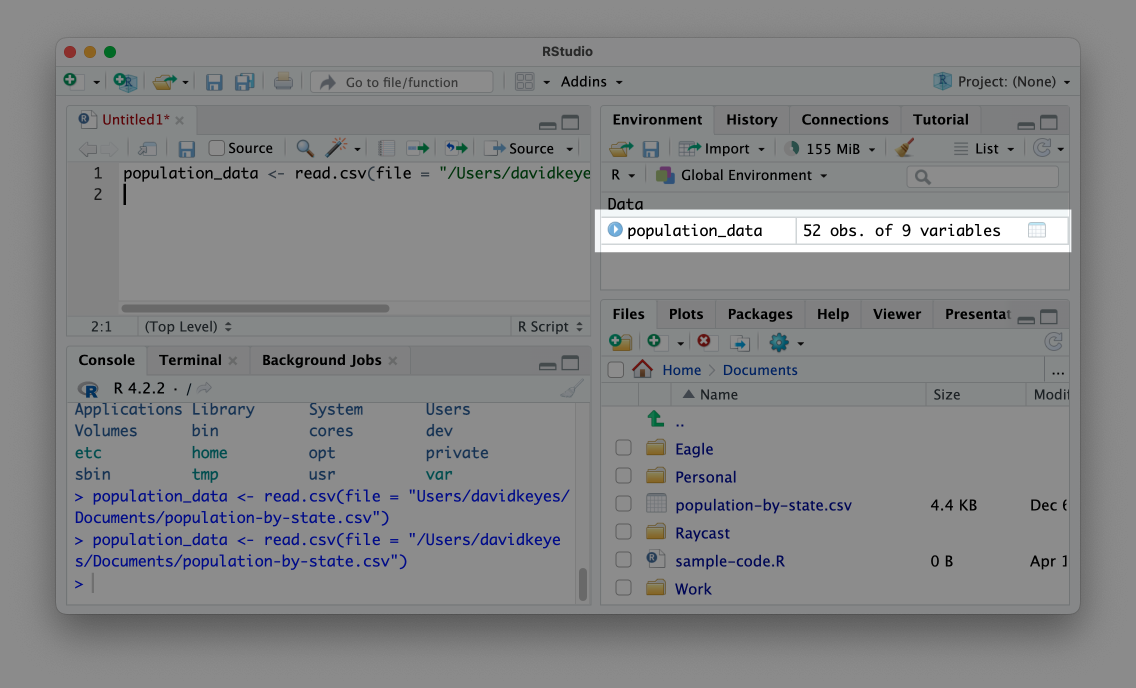
To save your data for reuse, you need to create an object. For the purposes of this discussion, an object is a data structure that is stored for later use. To create an object, update your data-importing syntax so it looks like this:

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

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

When you run this code, you should see population\_data in your environment pane, as shown in Figure 1-5.

[F01005.png]



* + - * 1. The population\_data object in the environment pane

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

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

Installing Packages

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

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

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

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

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

library(tidyverse)

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

At the top of the script, the line library(tidyverse) loads the tidyverse package. Then, the package’s read\_csv() function imports the data. Note the underscore (\_) in place of the period (.) in the function’s name; this differs from the base R function you used earlier. Using read\_csv() to import CSV files achieves the same goal of creating an object, however—in this case, one called population\_data\_2. Enter population\_data\_2 in the console, and you should see this output:

#> # A tibble: 52 × 9

#> rank State Pop Growth Pop2018 Pop2010

#> <dbl> <chr> <dbl> <dbl> <dbl> <dbl>

#> 1 1 California 39613493 0.0038 39461588 37319502

#> 2 2 Texas 29730311 0.0385 28628666 25241971

#> 3 3 Florida 21944577 0.0330 21244317 18845537

#> 4 4 New York 19299981 -0.0118 19530351 19399878

#> 5 5 Pennsylvania 12804123 0.0003 12800922 12711160

#> 6 6 Illinois 12569321 -0.0121 12723071 12840503

#> 7 7 Ohio 11714618 0.0033 11676341 11539336

#> 8 8 Georgia 10830007 0.0303 10511131 9711881

#> 9 9 North Carolina 10701022 0.0308 10381615 9574323

#> 10 10 Michigan 9992427 0.0008 9984072 9877510

#> # 42 more rows

#> # 3 more variables: growthSince2010 <dbl>, Percent <dbl>,

#> # density <dbl>

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

RStudio Projects

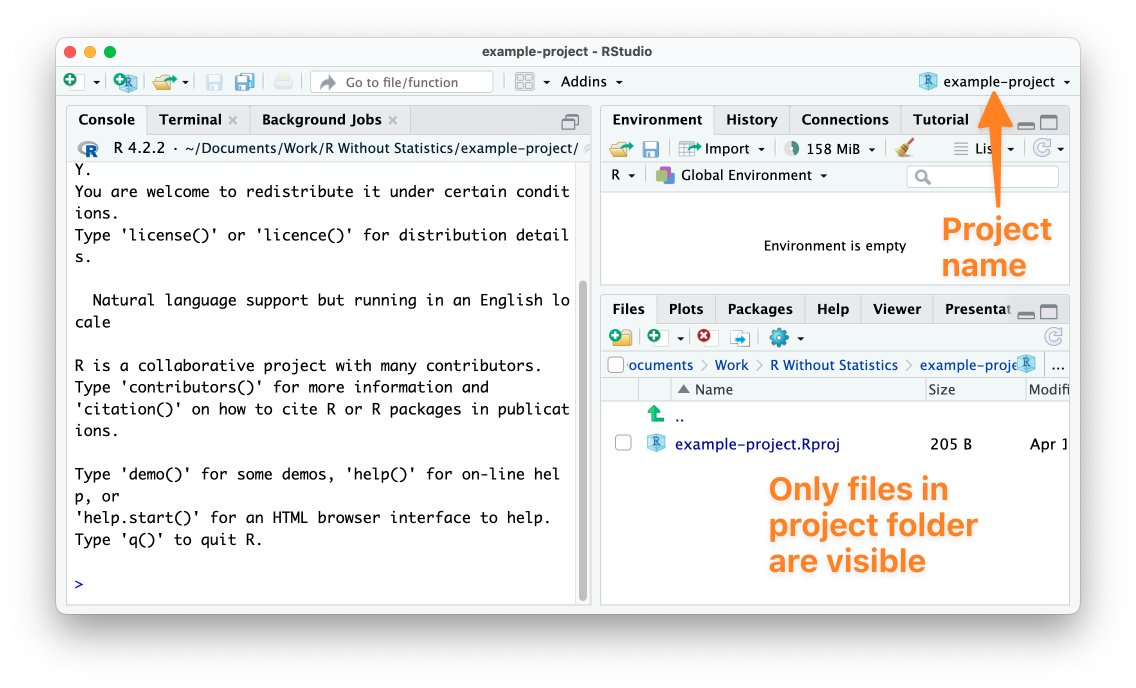
So far, you’ve imported a CSV file from your Documents folder. But because others won’t have this exact location on their computer, your code won’t work if they try to run it. One solution to this problem is an RStudio project.

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

To create a new RStudio project, go to 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. Next, choose a name for the new directory and where it should live. (Leave the checkboxes that ask about creating a Git repository and using renv unchecked; they’re for more advanced purposes.)

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

[F01006.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 so, you should see it in the RStudio files pane.

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

library(tidyverse)

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

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

Data Analysis with tidyverse

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

tidyverse Functions

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

summarize(.data = population\_data\_2, mean\_population = mean(Pop))

First, the code passes population\_data\_2 to the summarize() function’s .data argument to tell R to use that data frame to perform the calculation. Next, it creates a new variable called mean\_population and assigns it to the output of the mean() function introduced earlier. The mean() function runs on Pop, one of the variables in the population\_data\_2 data frame.

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

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

#> # A tibble: 1 × 1

#> mean\_population

#> <dbl>

#> 1 6433422.

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

Notice also that the summarize() function creates a totally new tibble from the original population\_data\_2 data frame. This is why the variables from population\_data\_2 are no longer present in the output.

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

The tidyverse Pipe

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

population\_data\_2 %>%

summarize(mean\_population = mean(Pop))

This code says, “Start with the population\_data\_2 data frame, then run the summarize() function on it, creating a variable called mean\_population by calculating the mean of the Pop variable.”

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

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

population\_data\_2 %>%

filter(rank <= 5) %>%

summarize(mean\_population = mean(Pop))

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

#> # A tibble: 1 × 1

#> mean\_population

#> <dbl>

#> 1 24678497

Using the pipe to combine functions lets you refine your data in multiple ways while keeping it readable and easy to understand. Indentation can also make your code more readable.

You’ve seen only a few functions for analysis at this point, but tidyverse has many more functions that enable you to do nearly anything you could hope to do with your data. Because of how useful tidyverse is, it will appear in every single piece of R code you write in this book.

Note R for Data Science, 2nd edition (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 the package’s many functions work.

Comments

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

# Calculate the mean population of the five largest states

population\_data\_2 %>%

filter(rank <= 5) %>%

summarize(mean\_population = mean(Pop))

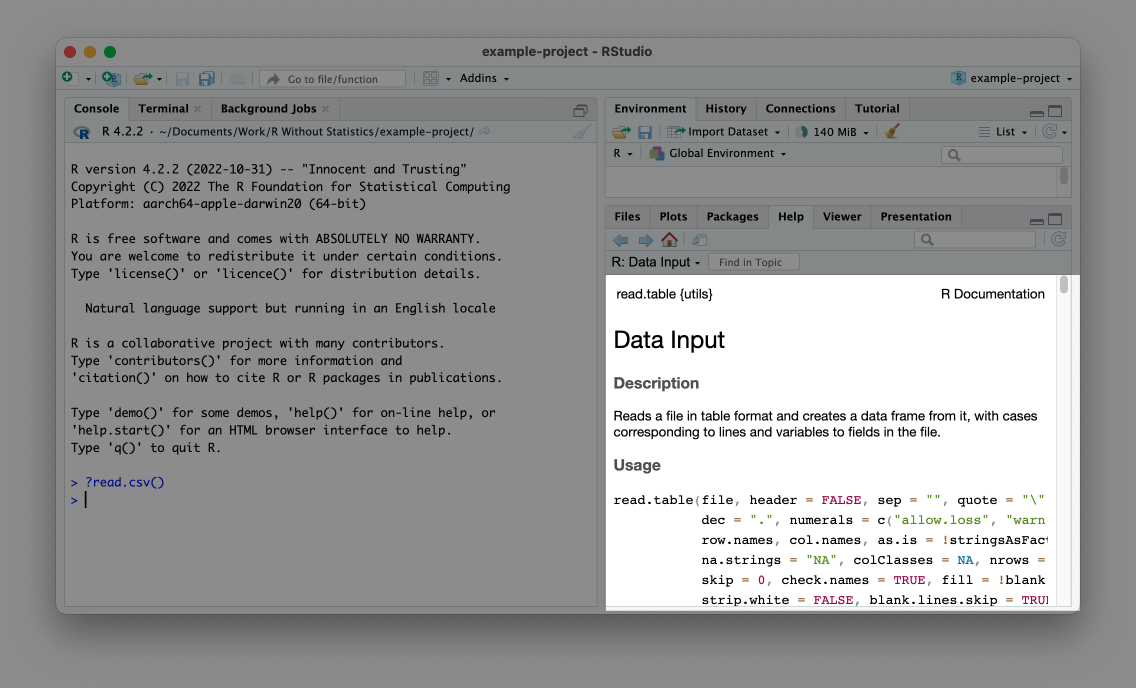
This comment will help others understand what is happening in the code, and it can also serve as a useful reminder for you if you haven’t worked on the code in a while. R knows to ignore any lines that begin with the hash mark instead of trying to run them.

How to Get Help

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

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

[F01007.png]



* + - * 1. The documentation for the read.csv() function

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

Note For additional guidance on reading documentation, I recommend the appendix of Kieran Healy’s book Data Visualization: A Practical Introduction (Princeton University Press, 2018). A free online version is available at https://socviz.co/appendix.html.

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

Conclusion

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

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

Further Resources

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

David Keyes, “Getting Started with R,” online course, accessed November 10, 2023, https://rfortherestofus.com/courses/getting-started.