



# BEGIN AT THE BEGINNING WITH R

Copyright ©2022 by SAGE Publications, Inc.

This work may not be reproduced or distributed in any form or by any means without express written permission of the publisher.

## LEARNING OBJECTIVES

Know how to install the R software package.

Gain familiarity with using the R command line.

Build and manipulate vectors in R.

If you are new to computers, programming, and/or data science, welcome to an exciting world that will open the door to the most powerful free data analytics tool ever created anywhere in the universe, no joke. On the other hand, if you are experienced with spreadsheets, statistical analysis, or accounting software you are probably thinking, as you start this chapter, that this book has gone off the deep end, never to return to sanity and all that is good and right in user interaction design. Both perspectives are reasonable. The R open source data analysis programming language is immensely powerful, flexible, and extensible (meaning that people can create new capabilities for it quite easily). At the same time, R is code-oriented, meaning that most of the work that one needs to perform is done through carefully crafted text instructions, many of which have very specific syntax (the punctuation and related rules for creating a command that works). Additionally, like many programming languages, R is not especially good at giving feedback or error messages that help the user to fix mistakes or figure out what is wrong when results look funny.

But there is a method to the madness here. One of the virtues of R as a teaching tool is that it hides very little. A successful user must fully understand what the data situation is or else the R commands will not work. With a spreadsheet, it is easy to type in a lot of numbers and a formula like =FORECAST and a result pops into a cell like magic, whether the results make any sense or not. With R you have to know your data, know what you can do with it, know how it must be transformed, and know how to check for trouble. Because R is a programming language, it also forces users to think about problems in terms of data objects, methods that can be applied to those objects, and procedures for applying those methods. These are important metaphors used in modern programming languages, and no data scientist can succeed without having at least a rudimentary understanding of how software is programmed, tested, and integrated into working systems.

Copyright ©2022 by SAGE Publications, Inc.

This work may not be reproduced or distributed in any form or by any means without express written permission of the publisher.

The extensibility of R means that new packages are being added all the time by volunteers: for example, R was among the first analysis programs to integrate capabilities for drawing data directly from Internet sources such as web pages and social media posts. You can be sure that, whatever the next big development is in the world of data, someone in the R community will start to develop a new package for R that will make use of it.

Finally, the lessons we can learn by working with R have nearly universal applicability to other programs and environments. If you have mastered R, it is a small step to get the hang of Python, another important data science language. Similarly, SAS<sup>®</sup> and SPSS<sup>®</sup>, two of the most widely used commercial statistical analysis programs, both have languages that will be easier to understand after you use R. Because R is open source, there are no licensing fees paid by schools, students, or teachers. As a result, it is possible to learn the most powerful data analysis system in the universe for free and take those lessons with you no matter where you go. It will take some patience though, so please hang in there!

## INSTALLING R

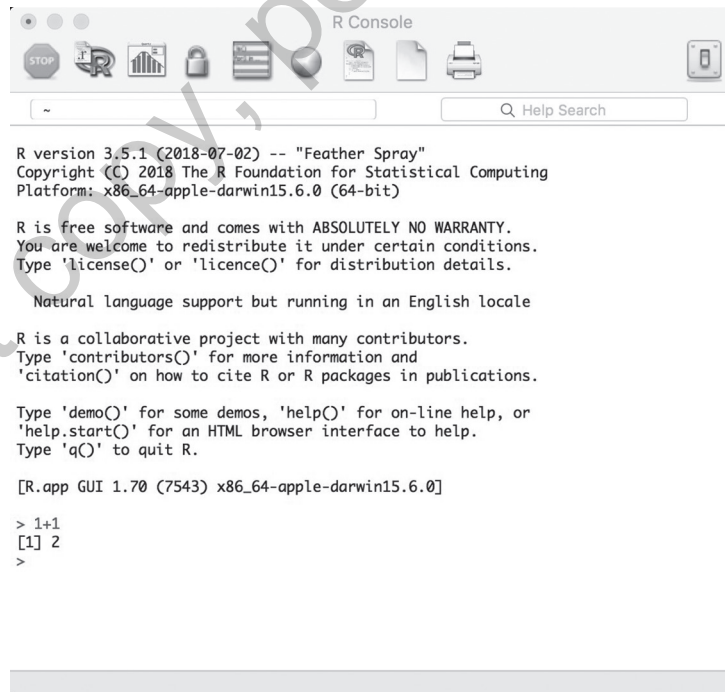
---

Let's get started. Obviously, you will need a computer. If you are working on a tablet device or smartphone, you might want to skip to the section on RStudio, because plain old R has not yet been reconfigured to work on tablet devices (there is a workaround for this that uses RStudio). There are a few experiments with web-based interfaces to R, like Jupyter Notebooks—but for now, we will focus on R and RStudio. If your computer has the Windows<sup>®</sup>, Mac-OS-X<sup>®</sup>, or a Linux operating system, there is a version of R waiting for you at <http://cran.r-project.org/>. Download and install your own copy. If you have difficulties with installing new software and you need some help, there is a wonderful little book by Thomas P. Hogan called *Bare-Bones R: A Brief Introductory Guide* (2009, Thousand Oaks, CA: SAGE) that you might want to buy or borrow from your library. There are lots of sites online that also give help with installing R, although many of them are not oriented toward the inexperienced user. We searched online using the term “help installing R” and got a few good hits. One site that was quite informative for installing R on Windows was [readthedocs.org](http://readthedocs.org), and you can try to access it at this TinyUrl: <http://tinyurl.com/872ngtt>. YouTube also has videos that provide brief tutorials for installing R. Try searching for “install R” in the YouTube search box. The rest of this

chapter assumes that you have installed R and can run it on your computer as shown in the first screenshot. Note that this screenshot is from the Mac version of R: if you are running Windows or Linux, your R screen could appear slightly different from this. If you are having trouble installing R and RStudio, you can also try <https://rstudio.cloud>, which is a web-based version of RStudio.

## USING R

The following screenshot shows a simple command to type that shows the most basic method of interaction with R. Notice near the bottom of the screenshot a greater than (>) symbol. This is the command prompt: When R is running and it is the active application on your desktop, if you type a command it appears after the > symbol. If you press the enter or return key, the command is sent to R for processing. When the processing is done, a result appears just under the >, if the command you used creates a result. When R is done processing, another command prompt (>) appears and R is ready for your next command. In the screenshot, the user has typed “1+1” and pressed the enter key. The formula 1+1 is used by elementary school students everywhere to insult each other’s math



```
R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (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.

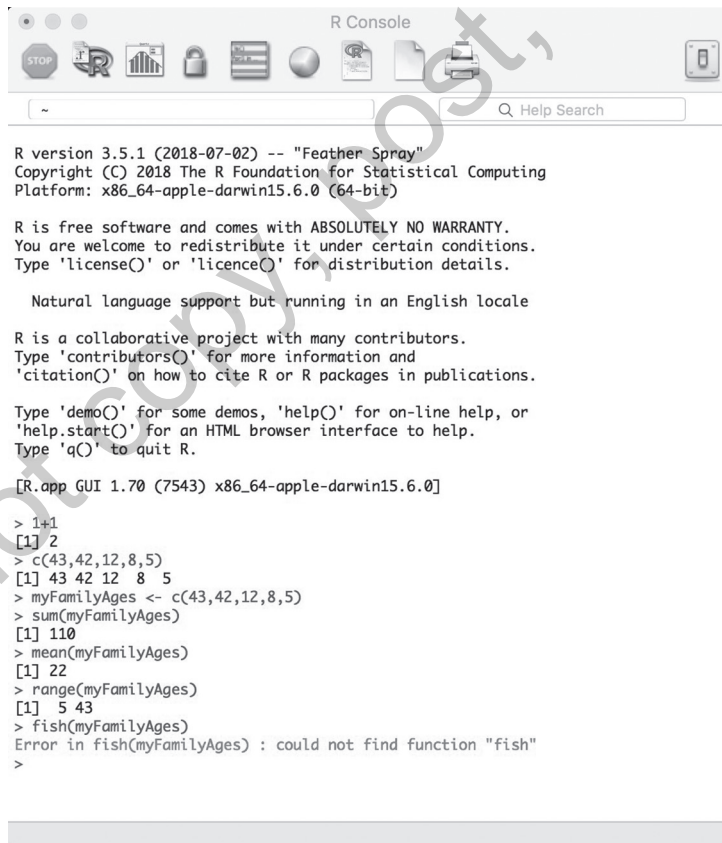
Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

[R.app GUI 1.70 (7543) x86_64-apple-darwin15.6.0]
> 1+1
[1] 2
>
```

skills, but R dutifully reports the result as 2. If you are a careful observer, you will notice that just before the 2 there is a 1 in square brackets, like this: [1]. That [1] is an index that helps to keep track of the results that R displays. Pretty pointless when only showing one result, but R likes to be consistent, so we will see quite a lot of those numbers in square brackets as we dig deeper.

## CREATING AND USING VECTORS

Here is a list of ages of family members: 43, 42, 12, 8, 5, for Dad, Mom, Sis, Bro, and their Dog, respectively. This is a list of items, all of the same mode (or type), namely an integer. They are integers because there are no decimal points and therefore nothing after the decimal point. We can create a vector of integers in R using the `c()` command. Take a look at the screen shot just below.



```

R Console
~
Q Help Search

R version 3.5.1 (2018-07-02) -- "Feather Spray"
Copyright (C) 2018 The R Foundation for Statistical Computing
Platform: x86_64-apple-darwin15.6.0 (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.

[R.app GUI 1.70 (7543) x86_64-apple-darwin15.6.0]
> 1+1
[1] 2
> c(43,42,12,8,5)
[1] 43 42 12 8 5
> myFamilyAges <- c(43,42,12,8,5)
> sum(myFamilyAges)
[1] 110
> mean(myFamilyAges)
[1] 22
> range(myFamilyAges)
[1] 5 43
> fish(myFamilyAges)
Error in fish(myFamilyAges) : could not find function "fish"
>

```

This is the last time that the whole screenshot from the R console will appear in the book. From here on out, we will just look at commands and output so we don't waste so much space on the page. The first command line creates a vector:

```
> c(43, 42, 12, 8, 5)
```

As you can see, when we show a short snippet of code we will make blue and bold what we type, and not blue and not bold what R is generating. So, in the above example, R generated the `>`, and then we typed `c(43, 42, 12, 8, 5)`. You don't need to type the `>` because R provides it whenever it is ready to receive new input. From now on in the book, there will be examples of R commands and output that are mixed together. The code will always be in blue and the output will be in black. In order to make it easier for you to reuse our code, we will be leaving out the `>` character from now on.

You might notice that on the following line in the screenshot, R dutifully reports the vector that you just typed. After the line number [1], we see the list 43, 42, 12, 8, and 5. This is because R echoes this list back to us, because we didn't ask it to store the vector anywhere. In the rest of the book, we will show that output from R as follows:

```
[1] 43, 42, 12, 8, 5
```

Combining these two lines, our R console snippet would look as follows:

```
c(43, 42, 12, 8, 5)
[1] 43, 42, 12, 8, 5
```

In contrast, the next command line is as follows:

```
myFamilyAges <- c(43, 42, 12, 8, 5)
```

We have typed in the same list of numbers, but this time we have assigned it, using the left-pointing arrow, into a storage area that we have named `myFamilyAges`. This time, R responds just with an empty command prompt. That's why the third command line requests a report of what `myFamilyAges` contains. This is a simple but very important tool.

Any time you want to know what is in a data object in R, just type the name of the object and R will report it back to you. In the next command, we begin to see the power of R:

```
sum(myFamilyAges)
[1] 110
```

This command asks R to add together all of the numbers in `myFamilyAges`, which turns out to be 110 (you can check it yourself with a calculator if you want). This is perhaps a weird thing to do with the ages of family members, but it shows how with a very short and simple command you can unleash quite a lot of processing on your data. In the next line (of the screenshot image), we ask for the mean (what non-data people call the average) of all of the ages and this turns out to be 22 years. The command right afterward, called `range`, shows the lowest and highest ages in the list. Finally, just for fun, we tried to issue the command `fish(myFamilyAges)`. Pretty much as you might expect, R does not contain a `fish()` function and so we received an error message to that effect. This shows another important principle for working with R. You can freely try things out at any time without fear of breaking anything. If R can't understand what you want to accomplish, or you haven't quite figured out how to do something, R will calmly respond with an error message and will not make any other changes until you give it a new command. The error messages from R are not always super helpful, but with some strategies that we will discuss in future chapters, you can break down the problem and figure out how to get R to do what you want.

Finally, it's important to remember that R is case sensitive. This means that `myFamilyAges` is different from `myFamilyages`. If we misspell the name of the data object by messing up the capitalization, we will get an error:

```
myFamilyages
Error: object 'myFamilyages' not found
```

Let's take stock for a moment. First, you should definitely try all of the commands noted above on your own computer. You can read about the commands in this book all you want, but you will learn a lot more if you actually try things out. Second, if you try a command that is shown in these pages and it does not work for some reason, you should try to

figure out why. Begin by checking your spelling and punctuation, because R is very picky about how commands are typed. Remember that capitalization matters in R: `myFamilyAges` is not the same as `myFamilyages`. If you verify that you have typed a command just as you see in the book and it still does not work, try to go online and look for some help. There's lots of help at <http://stackoverflow.com>, at <http://www.statmethods.net/>, and at many other web sites. If you can figure out what went wrong on your own, you will probably learn something very valuable about working with R. Third, you should take a moment to experiment with each new set of commands that you learn. For example, just using the commands discussed earlier in the chapter you could do this totally new thing:

```
myRange <- range(myFamilyAges)
myRange
```

What would happen if you did these two commands? What would you see? Think about how that worked and try to imagine some other experiments that you could try. The more you experiment on your own, the more you will learn. Some of the best stuff ever invented for computers was the result of just experimenting to see what was possible.

## SUBSETTING VECTORS

---

We can use the ability to access individual elements within a vector to select a subset of the elements in the vector. For example, we can select the third, the second, and the fifth elements of the `myFamilyAges` vector:

```
myFamilyAges[ c(3,2,5) ]
[1] 12 42 5
```

We can also define a sequential range of values (indices) and select those elements of the vector. The colon operator, as used in the following, creates the inclusive list of integers 3, 4, 5:

```
myFamilyAges[ c(3:5) ]
[1] 12 8 5
```



As you can see below, if the indices are negative, those vector elements are removed and whatever is left over is returned:

```
myFamilyAges[ c(-3:-5) ]
[1] 43 42
```

In addition to defining specific indices to select or remove elements from the vector, we can also tell R, via a list of TRUEs and FALSEs, which vector elements to include: either directly with a vector of TRUE and FALSE (once for each element in the vector) or by using a variable that contains a TRUE or FALSE for each element of the vector. As you can see, both approaches create the same result:

```
myFamilyAges[ c(TRUE, FALSE, TRUE, FALSE, FALSE) ]
[1] 43 12

selectedFamily <- c(TRUE, FALSE, TRUE, FALSE, FALSE)
myFamilyAges[ selectedFamily ]
[1] 43 12
```

Next, if we combine these concepts with a conditional evaluation, we can get a subset of a vector based on a conditional test of the data itself. In the example below, we test which family members are older than 21. The expression `myFamilyAges > 21` applies that conditional test to every element of `myFamilyAges` and produces a vector of TRUEs and FALSEs that is exactly the same length as `myFamilyAges`. Take the time to get a clear understanding of how this works, because it is a very powerful feature that we will use over and over again as we repair and transform data sets:

```
myFamilyAges > 21
[1] TRUE TRUE FALSE FALSE FALSE

selectedFamily <- myFamilyAges > 21
myFamilyAges[ selectedFamily ]
[1] 43 42
```

We can also do the conditional evaluation directly inside the square brackets. For example, we can identify all the family members that are a specific age (use two equal signs together to test equality in the conditional). We can also identify all the family members that are not a specific age (using the exclamation point “!” symbol). We can see that in R, the exclamation point in a conditional means “not.” Examine the output created by this conditional and explain in an R comment what happened.

```
myFamilyAges[ myFamilyAges == 12]
[1] 12

myFamilyAges[ myFamilyAges != 12]
[1] 43 42 8 5

myFamilyAges[ !(myFamilyAges == 12)]
[1] 43 42 8 5
```

Finally, we can also construct an expression that performs multiple conditional tests within the same line, such as all the family members older than 6 but younger than 20:

```
myFamilyAges[ myFamilyAges > 6 & myFamilyAges < 20 ]
[1] 12 8
```

Don't forget that one way to make sure that your conditional works correctly or to figure out the problem if it does not is to copy just the conditional expression and run it by itself to look at the map of TRUE and FALSE that it produces.

## THE COMMAND CONSOLE

---

For all of the commands shown above, we used the R console. Console is an old technology term that dates back to the days when computers were so big that they each occupied their own air-conditioned room. Within that room, there was often one master control station where a computer operator could do just about anything to control the giant computer by typing in commands. That station was known as the console. The term “console” is now used in many cases to refer to any interface where you can directly type

in commands. We've typed commands into the R console in an effort to learn about the R language as well as to illustrate some basic principles about data structures and statistics. If we really want to “do” data science, though, we can't sit around typing commands all day. First, it will become boring very fast. Second, whoever is paying us to be a data scientist will get suspicious when he or she notices that we are *retyping* some of the same commands we typed yesterday. Third, and perhaps most important, it is way too easy to make a mistake—to create what computer scientists refer to as a bug—if you are doing every little task by hand.

## USING AN INTEGRATED DEVELOPMENT ENVIRONMENT

---

To promote quality and efficiency in the R code that we create, one of our big goals within this book is to build something that is reusable: where we can do a few clicks or type a couple of things and unleash the power of many processing steps. Every software engineer knows that if you want to get serious about building something out of code, you must use an integrated development environment (IDE). Using an IDE, we can build these kinds of reusable pieces.

Starting in 2009, Joseph J. Allaire, a serial entrepreneur, software engineer, and the originator of some remarkable software products, began working with a small team to develop an open-source program to enhance the usability and power of R. As mentioned previously, R is an open-source program, meaning that the source code that is used to create a copy of R to run on a Mac, Windows, or Linux computer is available for all to inspect and modify. As with many open-source projects, there is an active community of developers who work on R, both on the basic program itself and on the many pieces and parts that can be added on to the basic program.

If you think of R as a piece of canvas lying on the table, RStudio is like an elegant picture frame where you can mount your canvas. R hangs in the middle of RStudio, and, like any good picture frame, RStudio enhances our appreciation of what is inside it. The IDE gives us the capability to open up the process of creation, to peer into the component parts when we need to, and to close the hood and hide them when we don't. Because we are working with data, we also need ways of inspecting the data, both its structure and its contents. As

you probably noticed, it gets tedious doing this at the R console, where almost every piece of output is a chunk of text and longer chunks scroll off the screen before you can see them. As an IDE for R, RStudio allows us to control and monitor our code, our data, our output, and our graphics in a way that supports the creation of reusable code sequences.

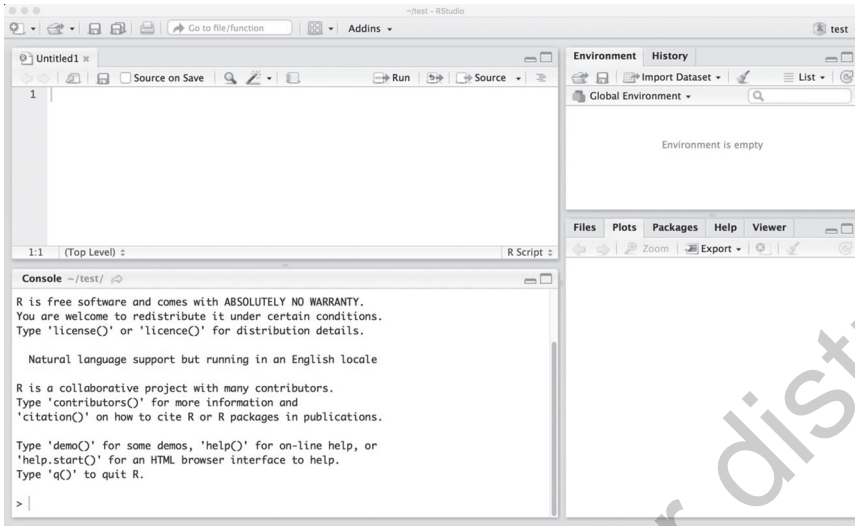
## INSTALLING RSTUDIO

---

Before we can get there, though, we have to have RStudio installed on a computer. Perhaps the most challenging aspect of installing RStudio is having to install R first, but if you've already done that, then installing RStudio should be a piece of cake. Make sure that you have the latest version of R installed before you begin with the installation of RStudio. There is ample documentation on the RStudio website, <http://www.rstudio.org/>, so if you follow the instructions there, you should have minimal difficulty. If you reach a page where you are asked to choose between installing RStudio server and installing RStudio as a desktop application on your computer, choose the latter. If you run into any difficulties or you just want some additional guidance about RStudio, you might want to have a look at the book entitled, *Getting Started with RStudio*, by John Verzani (2011, Sebastopol, CA: O'Reilly Media). The first chapter of that book has a general orientation to R and RStudio as well as a guide to installing and updating RStudio. There is also a YouTube video that introduces RStudio here: <http://www.youtube.com/watch?v=7sAmqkZ3Be8>

If you search for other YouTube videos, be aware that there is a disk recovery program as well as a music group that share the RStudio name: you will get a number of these videos if you search on "RStudio" without any other search terms.

Once you have installed RStudio, you can run it immediately in order to get started with the activities in the later parts of this chapter. Unlike other introductory materials, we will not walk through all of the different elements of the RStudio screen. Rather, as we need each feature we will highlight the new aspect of the application. When you run RStudio, you will see three or four sub-windows. Use the File menu to select New, and in the sub-menu for New, select "R Script." This should give you a screen that looks something like this:



## CREATING R SCRIPTS

Now let's use RStudio! In the lower-left-hand pane (another name for a sub-window) of RStudio, you will notice that we have a regular R console running. You can type commands into this console, just like we did earlier using plain old R:

Click in the console pane and type the following:

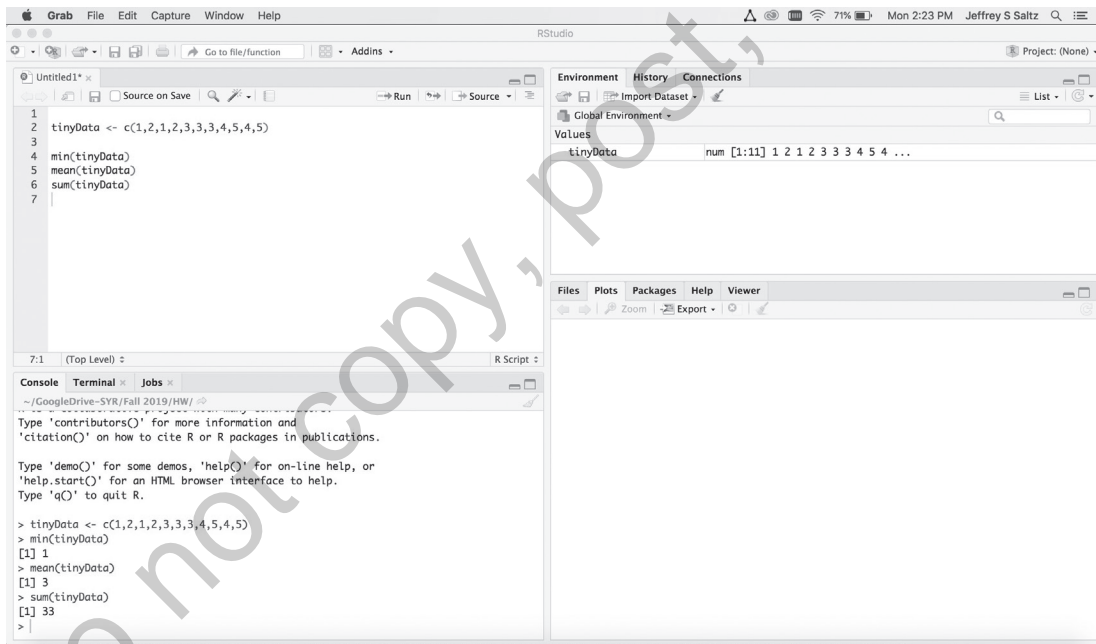
```
tinyData <- c(1,2,1,2,3,3,3,4,5,4,5)
mean(tinyData)
[1] 3
```

As you can see, this behaves the exact same way as just using the R console! However, it gets much more interesting if we use the upper-left-hand pane, which displays a blank space under the tab title Untitled1. This is the pane that contains your R source code file. Click on the source code pane (upper left pane), and then enter the following code:

```
tinyData <- c(1,2,1,2,3,3,3,4,5,4,5)

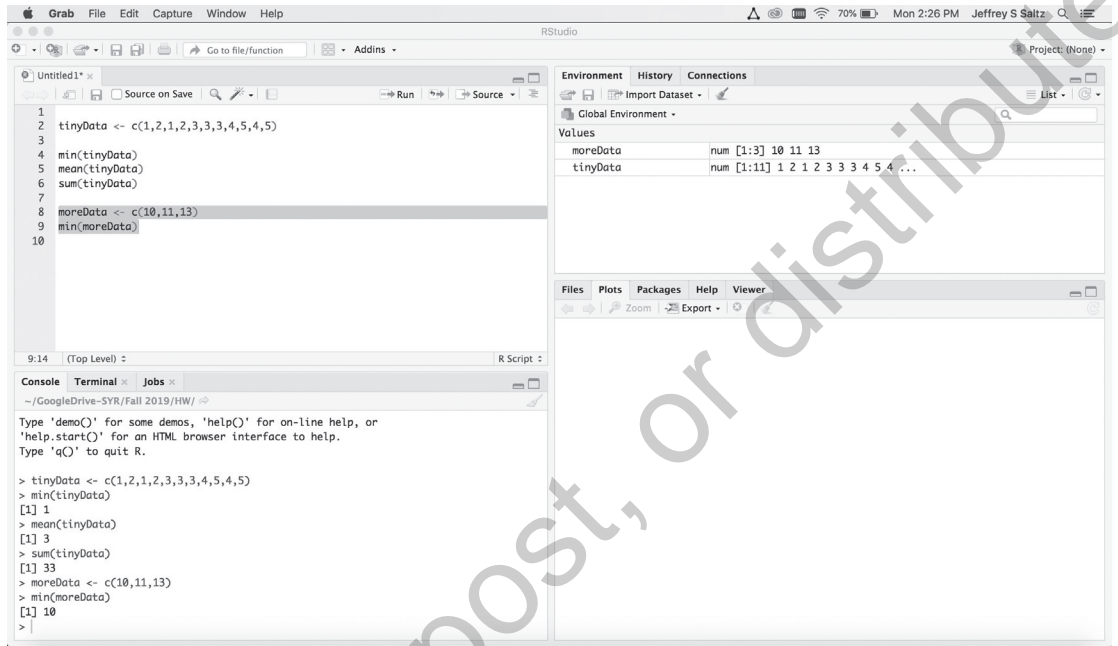
min(tinyData)
mean(tinyData)
sum(tinyData)
```

You can see, we are still writing R code, and rather than typing things at the “>” console prompt, we are just putting them in an R source code file that can be saved. Once we have the R source code, we can click on the Run button to run the commands that you wrote into the R script file (upper-left-hand pane). The Run button also has an icon with a little green arrow next to it—if you make the code window very small, sometimes all you will see is the icon! There are two ways to use the run button. You can highlight the piece of code that you want to run. You have to highlight carefully in order to choose the complete piece of code that you want to run. The other way to do it is that if you simply click in a line of code without highlighting anything and then press the Run button, you will run that whole line of code. Once you click Run, R will copy your code into the Console and the output (if any) will appear just below the code that was copied into the Console. R will then move the cursor to the next line of code in the source code file. So, clicking on Run again will execute the next line of code in your source code file.



Try it: it is just as if we had typed the commands into the console window, and that the output of min, mean, and sum commands show up in the R console pane. The advantage of this is that the code you just ran in the code window sticks around and can be saved. You can fix problems in the code, make copies of it, add comments, and do other things

that will help you be a more productive data scientist. You should also be able to see in the upper right pane (under the Environment tab) that there is a new data element `tinyData`. Your RStudio display should now look like this:



Let's practice highlighting code in the R source code window, and then clicking the “run” button. Here are two additional lines of code to type in: highlight and run:

```
moreData <- c(10,11,13)
min(moreData)

[1] 10
```

Note that in the upper right, the environment window now has the vector `moreData` as well as the vector `tinyData`. Another tab in the upper right is the History tab. This is useful to see the list of previous R commands we have executed. Although we have not yet discussed the lower-right window, we will use that window later to see the results of our visualizations.

To recap, this chapter provided a basic introduction to some basic R commands via the R Console and RStudio, an IDE for R. An IDE is useful for helping to build reusable components for handling data and conducting data analysis. From this point forward, we will use RStudio, rather than plain old R, in order to save and be able to reuse our work. Among other things, RStudio makes it easy to manage packages in R, and packages are the key to R's extensibility. In future chapters, we will be routinely using R packages to get access to specialized capabilities. These specialized capabilities come in the form of extra functions that are created by developers in the R community.

To summarize what we have discussed so far, you now know the following things about R (and about data):

- Installing and running R and RStudio on your computer.
- Typing commands on the R console.
- Using the `c()` function. Remember that `c` stands for combine, which just means to join things together. You can put a list of items inside the parentheses, separated by commas.
- A vector is the most basic form of data storage in R, and it consists of a list of items of the same type (such as numeric).
- A vector can be stored in a named location using the assignment arrow (a left-pointing arrow made of a dash and a less than symbol).
- You can get a report of the data object that is in any named location just by typing that name at the command line.
- Running a function, such as `mean()`, on a vector of numbers can transform them into something else. For example, `mean()` calculates the average, which is one of the most basic numeric summaries.
- `sum()`, `mean()`, and `range()` are all legal functions in R, whereas `fish()` is not.
- R is case sensitive.



## CASE STUDY: CALCULATING NPS

### Case Key Points:

- Define a vector that represents likelihood to recommend
- Calculate the number of promoters and detractors—  
Calculate net promoter score (NPS)

Let's take what we have learned in this chapter about using vectors, and write some R code that can calculate an overall NPS for a vector of data. As we discussed in the Case Overview (in the previous chapter), NPS is a measurement that many businesses use to quickly summarize the attitudes of a set of consumers toward the product or service that the business offers. To calculate NPS, we need a vector of numbers where each number represents a *likelihood to recommend* an answer to the question: “on a scale of 1 to 10, how likely are you to recommend this [product or service].” For our case study, we have asked flyers how likely they were to recommend the airline that provided their flight.

Below is the code to calculate NPS (in the source code window). There are several ideas to note with this first real code example. First, note the use of comments (starting with the “#” character). Second, note that `numPromoters` was calculated differently than `numDetractors`. This was done to show two alternative ways to do the same calculation. We could have used either approach for the number of detractors or the number of promoters. Finally, make sure you understand the trick in summing a vector of TRUE and FALSE values. The reason this works is that R represents TRUE as “1” and FALSE as “0.” As a result, calculating a sum on a vector of TRUE and FALSE values (often called Boolean values) results in a count of the number of TRUES.

```
# define a test vector
ltr <- c(9,8,3,9,7,8,9,6,7,8,9)

# what is the range of the ltr vector
range(ltr)

# create a new vector with just the promoters
# then calculate the length of the promoters vector
promoters <- ltr[ltr>8]
numPromoters <- length(promoters)
```

```
# calculate the number of detractors by summing the
# elements that are less than 7
detractorsTrueFalse <- ltr < 7
numDetractors <- sum(detractorsTrueFalse)

# calculate NPS, based on the length of the ltr vector
# and the number of promoters and detractors
total <- length(ltr)
nps <- (numPromoters/total - numDetractors/total)*100

# output NPS
nps
```

This code is in the source code editor, so highlight all of the code and then click the “Run” button. After selecting and running the code, you should see, in the console window, the range of the ltr vector and the actual NPS value:

```
range(ltr)
[1] 6 9

nps
[1] 18.18182
```

## Chapter Challenges

1. Use the `c()` function to add another family member's age onto the end of the `myFamilyAges` vector.
2. Use square brackets subsetting to show just the first element of the `myFamilyAges` vector.
3. Use square brackets subsetting together with the `c()` command to show just the odd numbered items from the `myFamilyAges` vector (i.e., just the first, third, and fifth items from this vector).
4. Create a conditional expression that outputs a set of TRUEs and FALSEs. The expression should show TRUE when an element of the `myFamilyAges` vector is equal to 12.
5. Using the code from the previous item, put an exclamation point in front of the conditional expression.
6. Use the conditional expression from the previous item within the square brackets to select those elements of the `myFamilyAges` vector that are not equal to 12.
7. **Power User:** Using the built-in Nile dataset, create a conditional expression that shows TRUE for every observation where the level of the Nile was over 900. Then use the `sum()` command to count up how many times the Nile dataset had observations higher than 900.

## Sources

[https://en.wikipedia.org/wiki/R\\_\(programming\\_language\)](https://en.wikipedia.org/wiki/R_(programming_language))  
[http://en.wikipedia.org/wiki/Joseph\\_J.\\_Allaire](http://en.wikipedia.org/wiki/Joseph_J._Allaire)  
<http://www.youtube.com/watch?v=7sAmqkZ3Be8>  
<https://www.rstudio.com/products/rstudio/features/>  
<http://a-little-book-of-r-for-biomedical-statistics.readthedocs.org/en/latest/src/installr.html>  
<http://cran.r-project.org/>  
[http://en.wikibooks.org/wiki/R\\_Programming](http://en.wikibooks.org/wiki/R_Programming)  
<http://stackoverflow.com>  
<http://www.statmethods.net/>

## R Functions Used in This Chapter

<code>c()</code>	Creates a vector.
<code>min()</code>	Finds the minimum number in a vector.
<code>mean()</code>	Finds the average for the entire vector.
<code>range()</code>	Finds the min and max values for the vector.
<code>sum()</code>	Finds the total for the entire vector.
<code>length()</code>	Finds the length (number of elements) of the vector.

This text includes access to datasets and select student resources. To learn more, visit [sagepub.com](http://sagepub.com)