PART II

From Excel to R

CHAPTER 6

First Steps with R for Excel Users

In Chapter 1 you learned how to conduct exploratory data analysis in Excel. You may recall from that chapter that John Tukey is credited with popularizing the practice of EDA. Tukey's approach to data inspired the development of several statistical programming languages, including S at the legendary Bell Laboratories. In turn, S inspired R. Developed in the early 1990s by Ross Ihaka and Robert Gentleman, the name is a play both on its derivation from S and its cofounders' first names. R is open source and maintained by the R Foundation for Statistical Computing. Because it was built primarily for statistical computation and graphics, it's most popular among researchers, statisticians, and data scientists.

R was developed specifically with statistical analysis in mind.

Downloading R

To get started, navigate to the R Project's website (*https://r-project.org*). Click the link at the top of the page to download R. You will be asked to choose a mirror from the Comprehensive R Archive Network (CRAN). This is a network of servers that distributes R source code, packages, and documentation. Choose a mirror near you to download R for your operating system.

Getting Started with RStudio

You've now installed R, but we will make one more download to optimize our coding experience. In Chapter 5, you learned that when software is open source, anyone is free to build on, distribute, or contribute to it. For example, vendors are welcome to offer an *integrated development environment* (IDE) to interact with the code. The RStudio IDE combines tools for code editing, graphics, documentation, and more under a single interface. This has become the predominant IDE for R programming in its decade or so on the market, with users building everything from interactive dashboards (Shiny) to research reports (R Markdown) with its suite of products.

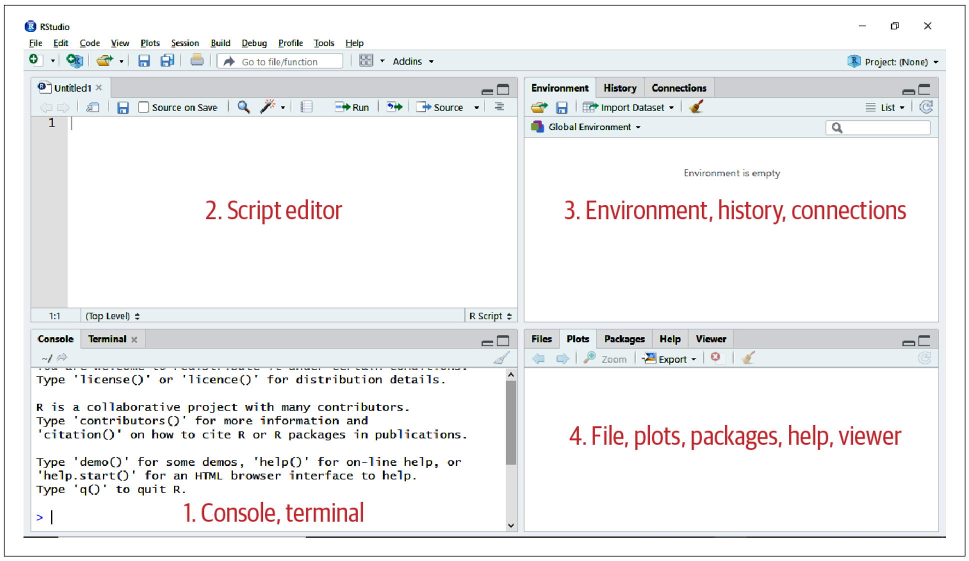
You may be wondering, *if RStudio is so great, why did we bother installing R?* These are in fact two distinct downloads: we downloaded R for the *code base*, and RStudio for an *IDE to work with the code*. This decoupling of applications may be unfamiliar to you as an Excel user, but it's quite common in the open source software world.

RStudio is a platform to *work with* R code, not the code base itself. First, download R from CRAN; then download RStudio.

To download RStudio, head to the download page (*https://oreil.ly/rfP1X*) of its website. You will see that RStudio is offered on a tiered pricing system; select the free RStudio Desktop. (RStudio is an excellent example of how to build a solid business on top of open source software.) You'll come to love RStudio, but it can be quite overwhelming at first with its many panes and features. To overcome this initial discomfort, we'll take a guided tour.

First, head to the home menu and select File → New File → R Script. You should now see something like Figure 6-1. There are lots of bells and whistles here; the idea of an IDE is to have all the tools needed for code development in one place. We'll cover the features in each of the four panes that you should know to get started.

Located in the lower left-hand corner of RStudio, the *console* is where commands are submitted to R to execute. Here you will see the > sign followed by a blinking cursor. You can type operations here and then press Enter to execute. Let's start with something very basic, like finding 1 + 1, as in Figure 6-2.


2. Script editor
3. Environment, history, connections
Environment is empty
4. File, plots, packages, help, viewer
1. Console, terminal

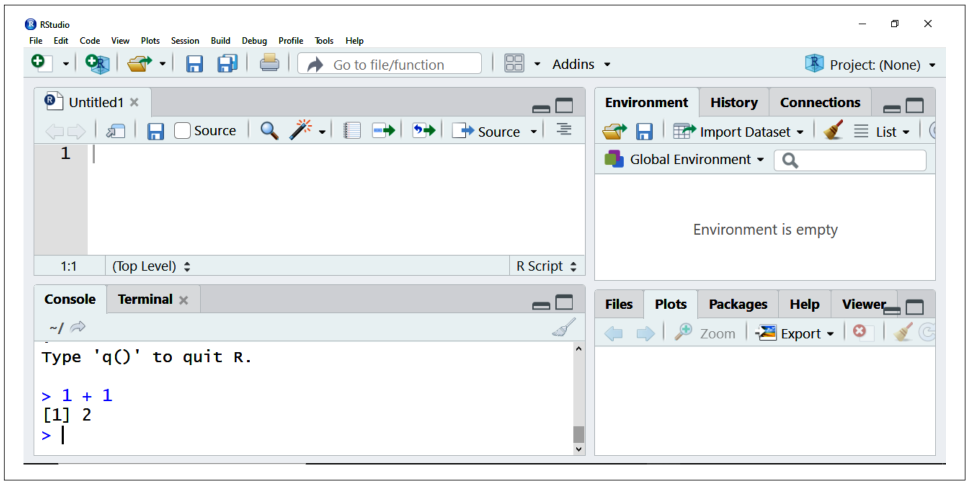
```
Type 'license()' or 'licence()' for distribution details.

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.

> |
```

Figure 6-1. The RStudio IDE



```
Type 'q()' to quit R.

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

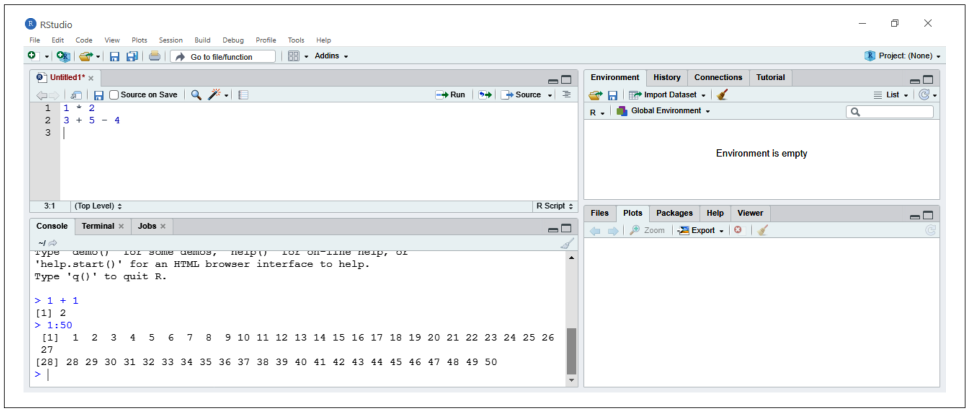
Figure 6-2. Coding in RStudio, starting with 1 + 1

You may have noticed that a `[1]` appears before your result of `2`. To understand what this means, type and execute `1:50` in the console. The `:` operator in R will produce all numbers in increments of 1 between a given range, akin to the fill handle in Excel. You should see something like this:

```
1:50
#> [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
#> [24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
#> [47] 47 48 49 50
```

These bracketed labels indicate the numeric position of the first value for each line in the output.

While you can continue to work from here, it's often a good idea to first write your commands in a *script*, and then send them to the console. This way you can save a long-term record of the code you ran. The script editor is found in the pane immediately above the console. Enter a couple of lines of simple arithmetic there, as in Figure 6-3.



```
1 * 2
3 + 5 - 4
```

```
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.

> 1 + 1
[1] 2
> 1:50
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26
 27
[28] 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50
> 
```

Figure 6-3. Working with the script editor in RStudio

Place your cursor in line 1, then hover over the icons at the top of the script editor until you find one that says "Run the current line or selection." Click that icon and two things will happen. First, the active line of code will be executed in the console. The cursor will also drop to the next line in the script editor. You can send multiple lines to the console at once by selecting them and clicking that icon. The keyboard shortcut for this operation is Ctrl + Enter for Windows, Cmd + Return for Mac. As an Excel user, you're probably a keyboard shortcut enthusiast; RStudio has an abundance of them, which can be viewed by selecting Tools → Keyboard Shortcuts Help.

Let's save our script. From the menu head to File → Save. Name the file *ch-6*. The file extension for R scripts is *.r*. The process of opening, saving, and closing R scripts may remind you of working with documents in a word processor; after all, they are both written records.

We'll now head to the lower-right pane. You will see five tabs here: Files, Plots, Packages, Help, Viewer. R provides plenty of help documentation, which can be viewed in this pane. For example, we can learn more about an R function with the ? operator.

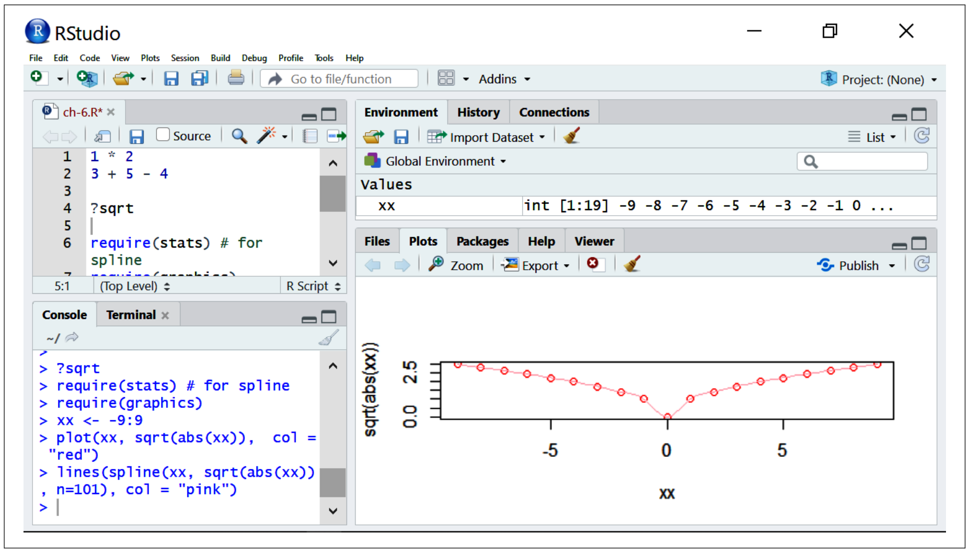
As an Excel user, you know all about functions such as `VLOOKUP()` or `SUMIF()`. Some R functions are quite similar to those of Excel; let's learn, for example, about R's square-root function, `sqrt()`. Enter the following code into a new line of your script and run it using either the menu icon or the keyboard shortcut:

```
?sqrt
```

A document titled "Miscellaneous Mathematical Functions" will appear in your Help window. This contains important information about the `sqrt()` function, the arguments it takes, and more. It also includes this example of the function in action:

```
require(stats) # for spline
require(graphics)
xx <- -9:9
plot(xx, sqrt(abs(xx)),  col = "red")
lines(spline(xx, sqrt(abs(xx)), n=101), col = "pink")
```

Don't worry about making sense of this code right now; just copy and paste the selection into your script, highlighting the complete selection, and run it. A plot will now appear as in Figure 6-4. I've resized my RStudio panes to make the plot larger. You will learn how to build R plots in Chapter 8.



```
1 * 2
3 + 5 - 4

?sqrt

require(stats) # for
spline
```

```
Values
  xx        int [1:19] -9 -8 -7 -6 -5 -4 -3 -2 -1 0 ...
```

```
> ?sqrt
> require(stats) # for spline
> require(graphics)
> xx <- -9:9
> plot(xx, sqrt(abs(xx)),  col =
 "red")
> lines(spline(xx, sqrt(abs(xx))
, n=101), col = "pink")
> 
```

Figure 6-4. Your first R plot

Now, look to your upper-right pane: Environment, History, Connections. The Environment tab lists something called `xx` next to what looks to be some set of integers. What is this? As it turns out, *you* created this with the code I told you to run blindly from the `sqrt()` documentation. In fact, much of what we do in R will focus around what is shown here: an *object*.

As you likely noticed, there are several panes, icons, and menu options we overlooked in this tour of RStudio. It's such a feature-rich IDE: don't be afraid to explore, experiment, and search-engine your way to learning more. But for now, you know enough about getting around in RStudio to begin learning R programming proper. You've already seen that R can be used as a fancy calculator. Table 6-1 lists some common arithmetic operators in R.

Table 6-1. Common arithmetic operators in R

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`^`	Exponent
`%%`	Modulo
`%/%`	Floor division

You may be less familiar with the last two operators in Table 6-1: the *modulo* returns the remainder of a division, and *floor division* rounds the division's result down to the nearest integer.

Like Excel, R follows the order of operations in arithmetic.

```
# Multiplication before addition
3 * 5 + 6
#> [1] 21

# Division before subtraction
2 / 2 - 7
#> [1] -6
```

What's the deal with the lines containing the hash (`#`) and text? Those are *cell comments* used to provide verbal instructions and reminders about the code. Comments help other users—and ourselves at a later date—understand and remember what the code is used for. R does not execute cell comments: this part of the script is for the programmer, not the computer. Though comments can be placed to the right of code, it's preferred to place them above:

```
1 * 2 # This comment is possible
#> [1] 2

# This comment is preferred
2 * 1
#> [1] 2
```

You don't need to use comments to explain *everything* about what your code is doing, but do explain your reasoning and assumptions. Think of it as, well, a commentary. I will continue to use comments in this book's examples where relevant and helpful.

Get into the habit of including comments to document your objectives, assumptions, and reasoning for writing the code.

As previously mentioned, functions are a large part of working in R, just as in Excel, and often look quite similar. For example, we can take the absolute value of –100:

```
# What is the absolute value of -100?
abs(-100)
#> [1] 100
```

However, there are some quite important differences for working with functions in R, as these errors indicate.

```
# These aren't going to work
ABS(-100)
#> Error in ABS(-100) : could not find function "ABS"
Abs(-100)
#> Error in Abs(-100) : could not find function "Abs"
```

In Excel, you can enter the `ABS()` function as lowercase `abs()` or proper case `Abs()` without a problem. In R, however, the `abs()` function *must* be lowercase. This is because R is *case-sensitive*. This is a major difference between Excel and R, and one that is sure to trip you up sooner or later.

R is a case-sensitive language: the `SQRT()` function is not the same as `sqrt()`.

Like in Excel, some R functions, like `sqrt()`, are meant to work with numbers; others, like `toupper()`, work with characters:

```
# Convert to upper case
toupper('I love R')
#> [1] "I LOVE R"
```

Let's look at another case where R behaves similarly to Excel, with one exception that will have huge implications: comparison operators. This is when we compare some relationship between two values, such as whether one is greater than the other.

```
# Is 3 greater than 4?
3 > 4
#> [1] FALSE
```

R will return a `TRUE` or `FALSE` as a result of any comparison operator, just as would Excel. Table 6-2 lists R's comparison operators.

Table 6-2. Comparison operators in R

Operator	Meaning
`>`	Greater than
`<`	Less than
`>=`	Greater than or equal to
`<=`	Less than or equal to
`!=`	Not equal to
`==`	Equal to

Most of these probably look familiar to you, except… did you catch that last one? That's correct, you do not check whether two values are equal to one another in R with one equal sign, but with *two*. This is because a single equal sign in R is used to *assign objects*.

Objects Versus Variables

Stored objects are also sometimes referred to as *variables* because of their ability to be overwritten and change values. However, we've already been referring to *variables* in the statistical sense throughout this book. To avoid this confusing terminology, we will continue to refer to "objects" in the programming sense and "variables" in the statistical.

If you're not quite sure what the big deal is yet, bear with me for another example. Let's assign the absolute value of –100 to an object; we'll call it `my_first_object`.

```
# Assigning an object in R
my_first_object = abs(-100)
```

You can think of an object as a shoebox that we are putting a piece of information into. By using the `=` operator, we've stored the result of `abs(-100)` in a shoebox called `my_first_object`. We can open this shoebox by *printing* it. In R, you can simply do this by running the object's name:

```
# Printing an object in R
my_first_object
#> [1] 100
```

Another way to assign objects in R is with the `<-` operator. In fact, this is usually preferred to = in part to avoid the confusion between it and ==. Try assigning another object using this operator, then printing it. The keyboard shortcut is Alt+- (Alt +minus) on Windows, and Option-- (Option-minus) on Mac. You can get creative with your functions and operations, like I did:

```
my_second_object <- sqrt(abs(-5 ^ 2))
my_second_object
#> [1] 5
```

Object names in R must start with a letter or dot and should contain only letters, numbers, underscores, and periods. There are also a few off-limit keywords. That leaves a lot of margin for "creative" object naming. But good object names are indicative of the data they store, similar to how the label on a shoebox signals what kind of shoe is inside.

R and Programming Style Guides

Some individuals and organizations have consolidated programming conventions into "style guides," just as a newspaper might have a style guide for writing. These style guides cover which assignment operators to use, how to name objects, and more. One such R style guide has been developed by Google and is available online (*https://oreil.ly/fAeJi*).

Objects can contain different types or *modes* of data, just as you might have different categories of shoeboxes. Table 6-3 lists some common data types.

Table 6-3. Common data types in R

Data type	Example
Character	`'R'`, `'Mount'`, `'Hello, world'`
Numeric	`6.2`, `4.13`, `3`
Integer	`3L`, `-1L`, `12L`
Logical	`TRUE`, `FALSE`, `T`, `F`

Let's create some objects of different modes. First, character data is often enclosed in single quotations for legibility, but double quotes also work and can be particularly helpful if you want to include a single quote as part of the input.

```
my_char <- 'Hello, world'
my_other_char <- "We're able to code R!"
```

Numbers can be represented as decimals or whole numbers:

```
my_num <- 3
my_other_num <- 3.21
```

However, whole numbers can also be stored as a distinct integer data type. The `L` included in the input stands for *literal*; this term comes from computer science and is used to refer to notations for fixed values:

```
my_int <- 12L
```

`T` and `F` will by default evaluate as logical data to `TRUE` and `FALSE`, respectively:

```
my_logical <- FALSE
my_other_logical <- F
```

We can use the `str()` function to learn about the *structure* of an object, such as its type and the information contained inside:

```
str(my_char)
#> chr "Hello, world"
str(my_num)
#> num 3
str(my_int)
#> int 12
str(my_logical)
#> logi FALSE
```

Once assigned, we are free to use these objects in additional operations:

```
# Is my_num equal to 5.5?
my_num == 5.5
#> [1] FALSE

# Number of characters in my_char
nchar(my_char)
#> [1] 12
```

We can even use objects as input in assigning other objects, or reassign them:

```
my_other_num <- 2.2
my_num <- my_num/my_other_num
my_num
#> [1] 1.363636
```

"So what?" you may be asking. "I work with a lot of data, so how is assigning each number to its own object going to help me?" Fortunately, you'll see in Chapter 7 that it's possible to combine multiple values into one object, much as you might do with ranges and worksheets in Excel. But before that, let's change gears for a moment to learn about packages.

Packages in R

Imagine if you weren't able to download applications on your smartphone. You could make phone calls, browse the internet, and jot notes to yourself—still pretty handy. But the real power of a smartphone comes from its applications, or apps. R ships much like a "factory-default" smartphone: it's still quite useful, and you could accomplish nearly anything necessary with it if you were forced to. But it's often more efficient to do the R equivalent of installing an app: *installing a package*.

The factory-default version of R is called "base R." Packages, the "apps" of R, are shareable units of code that include functions, datasets, documentation, and more. These packages are built on top of base R to improve functionality and add new features.

Earlier, you downloaded base R from CRAN. This network also hosts over 10,000 packages that have been contributed by R's vast user base and vetted by CRAN volunteers. This is your "app store" for R, and to repurpose the famous slogan, "There's a package for that." While it's possible to download packages elsewhere, it's best as a beginner to stick with what's hosted on CRAN. To install a package from CRAN, you can run `install.packages()`.

CRAN Task Views

It can be difficult for a newcomer to identify the right R packages for their needs. Fortunately, the CRAN team provides something like "curated playlists" of packages for given use cases with CRAN Task Views (*https://oreil.ly/q31wg*). These are bundles of packages meant to assist with everything from econometrics to genetics and provide a great landscape of helpful R packages. As you continue to learn the language, you'll get more comfortable locating and sizing up the right package for your requirements.

We'll be using packages in this book to help us with tasks like data manipulation and visualization. In particular, we'll be using the `tidyverse`, which is actually a *collection* of packages designed to be used together. To install this collection, run the following in the console:

```
install.packages('tidyverse')
```

You've just installed a number of helpful packages; one of which, `dplyr` (usually pronounced *d-plier*), includes a function `arrange()`. Try opening the documentation for this function and you'll receive an error:

```
?arrange
#> No documentation for ‘arrange’ in specified packages and libraries:
#> you could try ‘??arrange’
```

To understand why R can't find this `tidyverse` function, go back to the smartphone analogy: even though you've installed an app, you still need to open it to use it. Same with R: we've installed the package with `install.packages()`, but now we need to call it into our session with `library()`:

```
# Call the tidyverse into our session
library(tidyverse)
#> -- Attaching packages ------------------------- tidyverse  1.3.0 --
#> v ggplot2 3.3.2     v purrr   0.3.4
#> v tibble  3.0.3     v dplyr   1.0.2
#> v tidyr   1.1.2     v stringr 1.4.0
#> v readr   1.3.1     v forcats 0.5.0
#> -- Conflicts ----------------------------- tidyverse_conflicts() --
#> x dplyr::filter() masks stats::filter()
#> x dplyr::lag()    masks stats::lag()
```

The packages of `tidyverse` are now available for the rest of your R session; you can now run the example without error.

Packages are *installed* once, but *called* for each session.

Upgrading R, RStudio, and R Packages

RStudio, R packages, and R itself are constantly improving, so it's a good idea to occasionally check for updates. To update RStudio, navigate to the menu and select Help → Check for Updates. If you're due for an update, RStudio will guide you through the steps.

To update all packages from CRAN, you can run this function and follow the prompted steps:

```
update.packages()
```

You can also update packages from the RStudio menu by going to Tools → Check for Package Updates. An Update Packages menu will appear; select all of the packages that you wish to update. You can also install packages via the Tools menu.

Upgrading R itself is unfortunately more involved. If you are on a Windows computer, you can use the `updateR()` function from the package `installr` and follow its instructions:

```
# Update R for Windows
install.packages('installr')
library(installr)
updateR()
```

For Mac, return to the CRAN website (*https://cran.r-project.org*) to install the latest version of R.

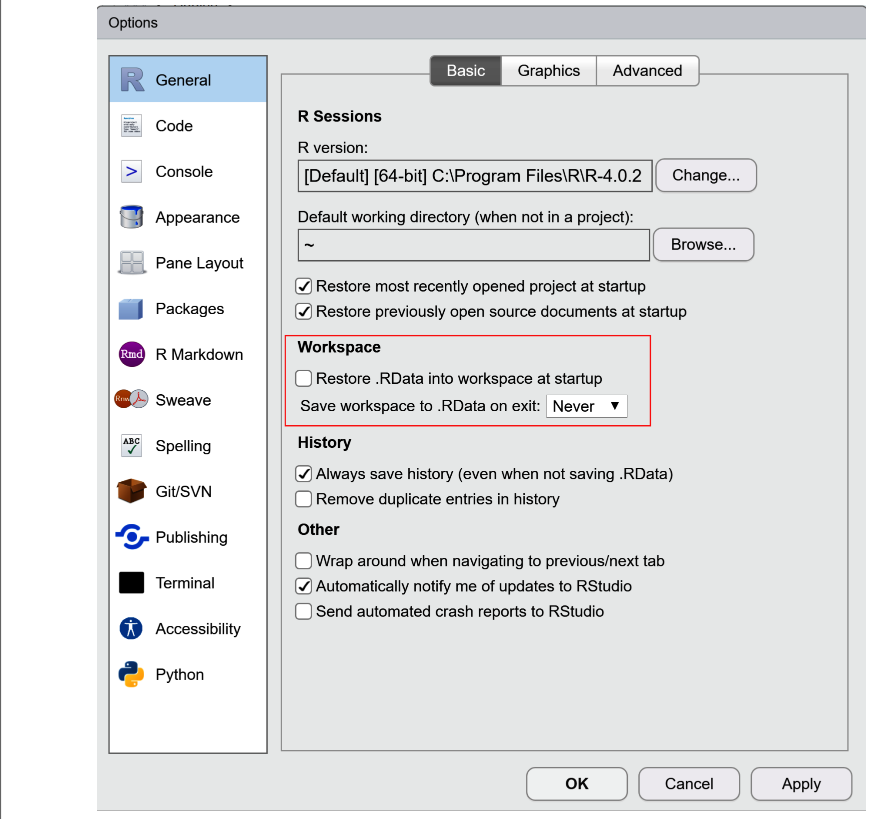
Conclusion

In this chapter, you learned how to work with objects and packages in R and got the hang of working with RStudio. You've learned a lot; I think it's time for a break. Go ahead and save your R script and close out of RStudio by selecting File → Quit Session. When you do so, you'll be asked: "Save workspace image to ~/.RData?" As a rule, *don't save your workspace image*. If you do, a copy of all saved objects will be saved so that they'll be available for your next session. While this *sounds* like a good idea, it can get cumbersome to store these objects and keep track of *why* you stored them in the first place.

Instead, rely on the R script itself to regenerate these objects in your next session. After all, the advantage of a programming language is that it's reproducible: no need to drag objects around with us if we can create them on demand.

Err on the side of *not* saving your workspace image; you should be able to re-create any objects from a previous session using your script.

To prevent RStudio from preserving your workspace between sessions, head to the home menu and go to Tools → Global Options. Under the General menu, change the two settings under Workspace as shown in Figure 6-5.



Options

General
Code
Console
Appearance
Pane Layout
Packages
R Markdown
Sweave
Spelling
Git/SVN
Publishing
Terminal
Accessibility
Python

Basic | Graphics | Advanced

R Sessions

R version:
[Default] [64-bit] C:\Program Files\R\R-4.0.2 | Change...

Default working directory (when not in a project):
~ | Browse...

☑ Restore most recently opened project at startup
☑ Restore previously open source documents at startup

Workspace

☐ Restore .RData into workspace at startup
Save workspace to .RData on exit: Never ▼

History

☑ Always save history (even when not saving .RData)
☐ Remove duplicate entries in history

Other

☐ Wrap around when navigating to previous/next tab
☑ Automatically notify me of updates to RStudio
☐ Send automated crash reports to RStudio

OK | Cancel | Apply

Figure 6-5. Customized workspace options in RStudio

Exercises

The following exercises provide additional practice and insight on working with objects, packages, and RStudio:

1. In addition to being a workhorse of a tool, RStudio provides endless appearance customizations. From the menu, select Tools → Global Options → Appearance and customize your editor's font and theme. For example, you may decide to use a "dark mode" theme.
2. Using a script in RStudio, do the following:
   - Assign the sum of 1 and 4 as `a`.
   - Assign the square root of `a` as `b`.
   - Assign `b` minus 1 as `d`.
   - What type of data is stored in `d`?
   - Is `d` greater than 2?
3. Install the `psych` package from CRAN, and load it into your session. Use comments to explain the differences between installing and loading a package.

Along with these exercises, I encourage you to begin using R immediately in your day-to-day work. For now, this may just involve using the application as a fancy calculator. But even this will help you get comfortable with using R and RStudio.

CHAPTER 7

Data Structures in R

Toward the end of Chapter 6 you learned how to work with packages in R. It's common to load any necessary packages at the beginning of a script so that there are no surprises about required downloads later on. In that spirit, we'll call in any packages needed for this chapter now. You may need to install some of these; if you need a refresher on doing that, look back to Chapter 6. I'll further explain these packages as we get to them.

```
# For importing and exploring data
library(tidyverse)

# For reading in Excel files
library(readxl)

# For descriptive statistics
library(psych)

# For writing data to Excel
library(writexl)
```

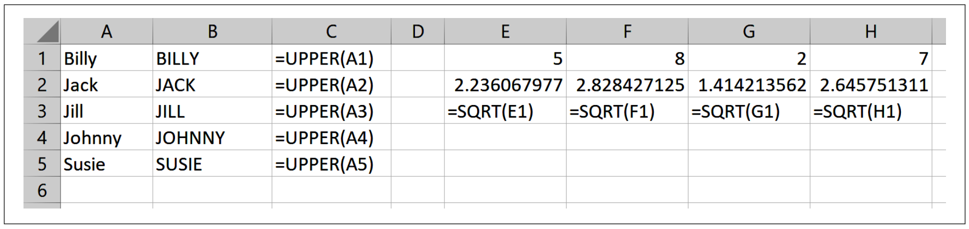
Vectors

In Chapter 6 you also learned about calling functions on data of different modes, and assigning data to objects:

```
my_number <- 8.2
sqrt(my_number)
#> [1] 2.863564

my_char <- 'Hello, world'
toupper(my_char)
#> [1] "HELLO, WORLD"
```

Chances are, you generally work with more than one piece of data at a time, so assigning each to its own object probably doesn't sound too useful. In Excel, you can place data into contiguous cells, called a *range*, and easily operate on that data. Figure 7-1 depicts some simple examples of operating on ranges of both numbers and text in Excel:



	A	B	C	D	E	F	G	H
1	Billy	BILLY	=UPPER(A1)		5	8	2	7
2	Jack	JACK	=UPPER(A2)		2.236067977	2.828427125	1.414213562	2.645751311
3	Jill	JILL	=UPPER(A3)		=SQRT(E1)	=SQRT(F1)	=SQRT(G1)	=SQRT(H1)
4	Johnny	JOHNNY	=UPPER(A4)					
5	Susie	SUSIE	=UPPER(A5)					
6								

Figure 7-1. Operating on ranges in Excel

Earlier I likened the *mode* of an object to a particular type of shoe in a shoebox. The *structure* of an object is the shape, size, and architecture of the shoebox itself. In fact, you've already been finding the structure of an R object with the `str()` function.

R contains several object structures: we can store and operate on a bit of data by placing it in a particular structure called a *vector*. Vectors are collections of one or more elements of data of the same type. Turns out we've already been using vectors, which we can confirm with the `is.vector()` function:

```
is.vector(my_number)
#> [1] TRUE
```

Though `my_number` is a vector, it only contains one element—sort of like a single cell in Excel. In R, we would say this vector has a length of 1:

```
length(my_number)
#> [1] 1
```

We can make a vector out of multiple elements, akin to an Excel range, with the `c()` function. This function is so called because it serves to *combine* multiple elements into a single vector. Let's try it:

```
my_numbers <- c(5, 8, 2, 7)
```

This object is indeed a vector, its data is numeric, and it has a length of 4:

```
is.vector(my_numbers)
#> [1] TRUE

str(my_numbers)
#> [1] num [1:4] 5 8 2 7

length(my_numbers)
#> [1] 4
```

Let's see what happens when we call a function on `my_numbers`:

```
sqrt(my_numbers)
#> [1] 2.236068 2.828427 1.414214 2.645751
```

Now we're getting somewhere. We could similarly operate on a character vector:

```
roster_names <- c('Jack', 'Jill', 'Billy', 'Susie', 'Johnny')
toupper(roster_names)
#> [1] "JACK"   "JILL"   "BILLY"  "SUSIE"  "JOHNNY"
```

By combining elements of data into vectors with the `c()` function, we were able to easily reproduce in R what was shown in Excel in Figure 7-1. What happens if elements of different types are assigned to the same vector? Let's give it a try:

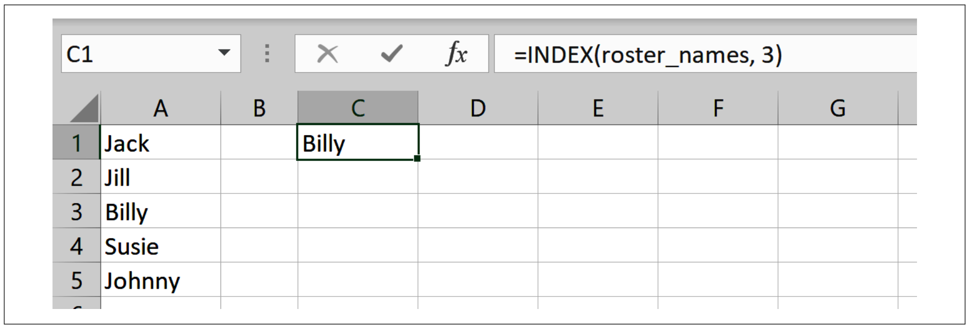
```
my_vec <- c('A', 2, 'C')
my_vec
#> [1] "A" "2" "C"

str(my_vec)
#> chr [1:3] "A" "2" "C"
```

R will *coerce* all elements to be of the same type so that they can be combined into a vector; for example, the numeric element `2` in the previous example was coerced into a character.

Indexing and Subsetting Vectors

In Excel, the `INDEX()` function serves to find the position of an element in a range. For example, I will use `INDEX()` in Figure 7-2 to extract the element in the third position of the named range `roster_names` (cells `A1:A5`):


C1
=INDEX(roster_names, 3)

	A	B	C	D	E	F	G
1	Jack		Billy				
2	Jill						
3	Billy						
4	Susie						
5	Johnny						

Figure 7-2. The `INDEX()` function on an Excel range

We can similarly index a vector in R by affixing the desired index position inside brackets to the object name:

```
# Get third element of roster_names vector
roster_names[3]
#> [1] "Billy"
```

Using this same notation, it's possible to select multiple elements by their index number, which we'll call *subsetting*. Let's again use the `:` operator to pull all elements between position 1 and 3:

```
# Get first through third elements
roster_names[1:3]
#> [1] "Jack"  "Jill"  "Billy"
```

It's possible to use functions here, too. Remember `length()`? We can use it to get everything through the last element in a vector:

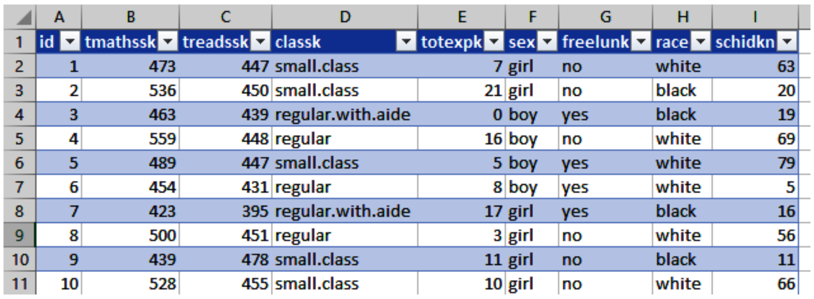
```
# Get second through last elements
roster_names[2:length(roster_names)]
#> [1] "Jill"   "Billy"  "Susie"  "Johnny"
```

We could even use the `c()` function to index by a vector of nonconsecutive elements:

```
# Get second and fifth elements
roster_names[c(2, 5)]
#> [1] "Jill"   "Johnny"
```

From Excel Tables to R Data Frames

"This is all well and good," you may be thinking, "but I don't just work with small ranges like these. What about whole data *tables*?" After all, in Chapter 1 you learned all about the importance of arranging data into variables and observations, such as the *star* data shown in Figure 7-3. This is an example of a *two-dimensional* data structure.



	A	B	C	D	E	F	G	H	I
1	id	tmathssk	treadssk	classk	totexpk	sex	freelunk	race	schidkn
2	1	473	447	small.class	7	girl	no	white	63
3	2	536	450	small.class	21	girl	no	black	20
4	3	463	439	regular.with.aide	0	boy	yes	black	19
5	4	559	448	regular	16	boy	no	white	69
6	5	489	447	small.class	5	boy	yes	white	79
7	6	454	431	regular	8	boy	yes	white	5
8	7	423	395	regular.with.aide	17	girl	yes	black	16
9	8	500	451	regular	3	girl	no	white	56
10	9	439	478	small.class	11	girl	no	black	11
11	10	528	455	small.class	10	girl	no	white	66

Figure 7-3. A two-dimensional data structure in Excel

Whereas R's vector is one-dimensional, the *data frame* allows for storing data in both rows *and* columns. This makes the data frame the R equivalent of an Excel table. Put formally, a data frame is a two-dimensional data structure where records in each column are of the same mode and all columns are of the same length. In R, like Excel, it's typical to assign each column a label or name.

We can make a data frame from scratch with the `data.frame()` function. Let's build and then print a data frame called `roster`:

```
roster <- data.frame(
   name = c('Jack', 'Jill', 'Billy', 'Susie', 'Johnny'),
   height = c(72, 65, 68, 69, 66),
   injured = c(FALSE, TRUE, FALSE, FALSE, TRUE))

roster
#>     name height injured
#> 1   Jack     72   FALSE
#> 2   Jill     65    TRUE
#> 3  Billy     68   FALSE
#> 4  Susie     69   FALSE
#> 5 Johnny     66    TRUE
```

We've used the `c()` function before to combine elements into a vector. And indeed, a data frame can be thought of as a *collection of vectors* of equal length. At three variables and five observations, `roster` is a pretty miniscule data frame. Fortunately, a data frame doesn't always have to be built from scratch like this. For instance, R comes installed with many datasets. You can view a listing of them with this function:

```
data()
```

A menu labeled "R data sets" will appear as a new window in your scripts pane. Many, but not all, of these datasets are structured as data frames. For example, you may have encountered the famous *iris* dataset before; this is available out of the box in R.

Just like with any object, it's possible to print *iris*; however, this will quickly overwhelm your console with 150 rows of data. (Imagine the problem compounded to thousands or millions of rows.) It's more common instead to print just the first few rows with the `head()` function:

```
head(iris)
#> Sepal.Length Sepal.Width Petal.Length Petal.Width Species
#> 1         5.1         3.5          1.4         0.2  setosa
#> 2         4.9         3.0          1.4         0.2  setosa
#> 3         4.7         3.2          1.3         0.2  setosa
#> 4         4.6         3.1          1.5         0.2  setosa
#> 5         5.0         3.6          1.4         0.2  setosa
#> 6         5.4         3.9          1.7         0.4  setosa
```

We can confirm that `iris` is indeed a data frame with `is.data.frame()`:

```
is.data.frame(iris)
#> [1] TRUE
```

Another way to get to know our new dataset besides printing it is with the `str()` function:

```
str(iris)
#> 'data.frame':	150 obs. of  5 variables:
#> $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
#> $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
#> $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
#> $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
#> $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 ...
```

The output returns the size of the data frame and some information about its columns. You'll see that four of them are numeric. The last, *Species*, is a *factor*. Factors are a special way to store variables that take on a limited number of values. They are especially helpful for storing *categorical* variables: in fact, you'll see that *Species* is described as having three *levels*, which is a term we've used statistically in describing categorical variables.

Though outside the scope of this book, factors carry many benefits for working with categorical variables, such as offering more memory-efficient storage. To learn more about factors, check out R's help documentation for the `factor()` function. (This can be done with the `?` operator.) The `tidyverse` also includes `forcats` as a core package to assist in working with factors.

In addition to the datasets that are preloaded with R, many packages include their own data. You can also find out about them with the `data()` function. Let's see if the `psych` package includes any datasets:

```
data(package = 'psych')
```

The "R data sets" menu will again launch in a new window; this time, an additional section called "Data sets in package `psych`" will appear. One of these datasets is called `sat.act`. To make this dataset available to our R session, we can again use the `data()` function. It's now an assigned R object that you can find in your Environment menu and use like any other object; let's confirm it's a data frame:

```
data('sat.act')
str(sat.act)
#> 'data.frame':	700 obs. of  6 variables:
#> $ gender   : int  2 2 2 1 1 1 2 1 2 2 ...
#> $ education: int  3 3 3 4 2 5 5 3 4 5 ...
#> $ age      : int  19 23 20 27 33 26 30 19 23 40 ...
#> $ ACT      : int  24 35 21 26 31 28 36 22 22 35 ...
#> $ SATV     : int  500 600 480 550 600 640 610 520 400 730 ...
#> $ SATQ     : int  500 500 470 520 550 640 500 560 600 800 ...
```

Importing Data in R

When working in Excel, it's common to store, analyze, and present data all within the same workbook. By contrast, it's uncommon to store data from inside an R script. Generally, data will be imported from external sources, ranging from text files and databases to web pages and application programming interfaces (APIs) to images and audio, and only then analyzed in R. Results of the analysis are then frequently exported to still different sources. Let's start this process by reading data from, not surprisingly, Excel workbooks (file extension *.xlsx*), and comma-separated value files (file extension *.csv*).

Base R Versus the tidyverse

In Chapter 6 you learned about the relationship between base R and R packages. Although packages can assist in doing things that would be quite difficult to do in base R, they sometimes offer alternative ways to do the same thing. For example, base R does include functions for reading *.csv* files (but not Excel files). It also includes options for plotting. We'll be using features of the `tidyverse` for these and other data needs. Depending on what you're looking to do, there's nothing wrong with the base R counterparts. I've decided to focus on `tidyverse` tools here because its syntax is more intelligible to Excel users.

To import data in R, it's important to understand how file paths and directories work. Each time you use the program, you're working from a "home base" on your computer, or a *working directory*. Any files you refer to from R, such as when you import a dataset, are assumed to be located relative to that working directory. The `getwd()` function prints the working directory's file path. If you are on Windows, you will see a result similar to this:

```
getwd()
#> [1] "C:/Users/User/Documents"
```

For Mac, it will look something like this:

```
getwd()
#> [1] "/Users/user"
```

R has a global default working directory, which is the same at each session startup. I'm assuming that you are running files from a downloaded or cloned copy of the book's companion repository, and that you are also working from an R script in that same folder. In that case, you're best off setting the working directory to this folder, which can be done with the `setwd()` function. If you're not used to working with file paths, it can be tricky to fill this out correctly; fortunately, RStudio includes a menu-driven approach for doing it.

To change your working directory to the same folder as your current R script, go to Session → Set Working Directory → To Source File Location. You should see the results of the `setwd()` function appear in the console. Try running `getwd()` again; you'll see that you are now in a different working directory.

Now that we've established the working directory, let's practice interacting with files relative to that directory. I have placed a *test-file.csv* file in the main folder of the book repository. We can use the `file.exists()` function to check whether we can successfully locate it:

```
file.exists('test-file.csv')
#> [1] TRUE
```

I have also placed a copy of this file in the *test-folder* subfolder of the repository. This time, we'll need to specify which subfolder to look in:

```
file.exists('test-folder/test-file.csv')
#> [1] TRUE
```

What happens if we need to go *up* a folder? Try placing a copy of *test-file* in whatever folder is one above your current directory. We can use `..` to tell R to look one folder up:

```
file.exists('../test-file.csv')
#> [1] TRUE
```

RStudio Projects

In the book repository, you will find a file called *aia-book.Rproj*. This is an RStudio project file. A project is a great way to preserve your work; for example, the project will maintain the configuration of windows and files that you had open in RStudio when you left. In addition, the project will automatically set your working directory to the *project* directory, so that you won't need a hardcoded `setwd()` for each script. When you work with R in this repository, then, consider doing so via the *.Rproj* file. You can then open any file via the Files pane in the lower right pane of RStudio.

Now that you have the hang of locating files in R, let's actually read some in. The book repository contains a *datasets* folder (*https://oreil.ly/wtneb*), under which is a *star* subfolder. This contains, among other things, two files: *districts.csv* and *star.xlsx*.

To read in *.csv* files, we can use the `read_csv()` function from `readr`. This package is part of the `tidyverse` collection, so we don't need to install or load anything new. We will pass the location of the file into the function. (Do you see now why understanding working directories and file paths was helpful?)

```
read_csv('datasets/star/districts.csv')
#>-- Column specification --------------------------
```

```
#> cols(
#>  schidkn = col_double(),
#>  school_name = col_character(),
#>  county = col_character()
#> )
#>
#> # A tibble: 89 x 3
#>   schidkn school_name     county
#>     <dbl> <chr>           <chr>
#> 1       1 Rosalia         New Liberty
#> 2       2 Montgomeryville Topton
#> 3       3 Davy            Wahpeton
#> 4       4 Steelton        Palestine
#> 5       5 Bonifay         Reddell
#> 6       6 Tolchester      Sattley
#> 7       7 Cahokia         Sattley
#> 8       8 Plattsmouth     Sugar Mountain
#> 9       9 Bainbridge      Manteca
#>10      10 Bull Run        Manteca
#> # ... with 79 more rows
```

RStudio's Import Dataset Wizard

If you are struggling to import a dataset, try RStudio's menu-driven data importer by heading to File → Import Dataset. You will be presented with a series of options to walk you through the process, including the ability to navigate to the source file via your computer's file explorer.

This results in a fair amount of output. First, our columns are specified, and we're told which functions were used to parse the data into R. Next, the first few rows of the data are listed, as a *tibble*. This is a modernized take on the data frame. It's still a data frame, and behaves mostly like a data frame, with some modifications to make it easier to work with, especially within the `tidyverse`.

Although we were able to read our data into R, we won't be able to do much with it unless we assign it to an object:

```
districts <- read_csv('datasets/star/districts.csv')
```

Among its many benefits, one nice thing about the tibble is we can print it without having to worry about overwhelming the console output; the first 10 rows only are printed:

```
districts
#> # A tibble: 89 x 3
#>   schidkn school_name     county
#>     <dbl> <chr>           <chr>
#> 1       1 Rosalia         New Liberty
#> 2       2 Montgomeryville Topton
#> 3       3 Davy           Wahpeton
#> 4       4 Steelton       Palestine
#> 5       5 Bonifay        Reddell
#> 6       6 Tolchester     Sattley
#> 7       7 Cahokia        Sattley
#> 8       8 Plattsmouth    Sugar Mountain
#> 9       9 Bainbridge     Manteca
#> 10      10 Bull Run       Manteca
#> # ... with 79 more rows
```

`readr` does not include a way to import Excel workbooks; we will instead use the `readxl` package. While it is part of the `tidyverse`, this package does not load with the core suite of packages like `readr` does, which is why we imported it separately at the beginning of the chapter.

We'll use the `read_xlsx()` function to similarly import *star.xlsx* as a tibble:

```
star <- read_xlsx('datasets/star/star.xlsx')
head(star)
#> # A tibble: 6 x 8
#>  tmathssk treadssk classk       totexpk sex   freelunk race  schidkn
#>     <dbl>    <dbl> <chr>          <dbl> <chr> <chr>    <chr>   <dbl>
#> 1      473      447 small.class        7 girl  no       white      63
#> 2      536      450 small.class       21 girl  no       black      20
#> 3      463      439 regular.wit~       0 boy   yes      black      19
#> 4      559      448 regular           16 boy   no       white      69
#> 5      489      447 small.class        5 boy   yes      white      79
#> 6      454      431 regular            8 boy   yes      white       5
```

There's more you can do with `readxl`, such as reading in *.xls* or *.xlsm* files and reading in specific worksheets or ranges of a workbook. To learn more, check out the package's documentation (*https://oreil.ly/kuZPE*).

Exploring a Data Frame

Earlier you learned about `head()` and `str()` to size up a data frame. Here are a few more helpful functions. First, `View()` is a function from RStudio whose output will be very welcome to you as an Excel user:

```
View(star)
```

After calling this function, a spreadsheet-like viewer will appear in a new window in your Scripts pane. You can sort, filter, and explore your dataset much like you would in Excel. However, as the function implies, it's for viewing *only*. You cannot make changes to the data frame from this window.

The `glimpse()` function is another way to print several records of the data frame, along with its column names and types. This function comes from `dplyr`, which is part of the `tidyverse`. We will lean heavily on `dplyr` in later chapters to manipulate data.

```
glimpse(star)
#> Rows: 5,748
#> Columns: 8
#> $ tmathssk <dbl> 473, 536, 463, 559, 489,...
#> $ treadssk <dbl> 447, 450, 439, 448, 447,...
#> $ classk   <chr> "small.class", "small.cl...
#> $ totexpk  <dbl> 7, 21, 0, 16, 5, 8, 17, ...
#> $ sex      <chr> "girl", "girl", "boy", "...
#> $ freelunk <chr> "no", "no", "yes", "no",...
#> $ race     <chr> "white", "black", "black...
#> $ schidkn  <dbl> 63, 20, 19, 69, 79, 5, 1...
```

There's also the `summary()` function from base R, which produces summaries of various R objects. When a data frame is passed into `summary()`, some basic descriptive statistics are provided:

```
summary(star)
#>     tmathssk        treadssk        classk             totexpk
#>  Min.   :320.0   Min.   :315.0   Length:5748        Min.   : 0.000
#>  1st Qu.:454.0   1st Qu.:414.0   Class :character   1st Qu.: 5.000
#>  Median :484.0   Median :433.0   Mode  :character   Median : 9.000
#>  Mean   :485.6   Mean   :436.7                      Mean   : 9.307
#>  3rd Qu.:513.0   3rd Qu.:453.0                      3rd Qu.:13.000
#>  Max.   :626.0   Max.   :627.0                      Max.   :27.000
#>      sex              freelunk            race
#>  Length:5748        Length:5748        Length:5748
#>  Class :character   Class :character   Class :character
#>  Mode  :character   Mode  :character   Mode  :character
#>     schidkn
#>  Min.   : 1.00
#>  1st Qu.:20.00
#>  Median :39.00
#>  Mean   :39.84
#>  3rd Qu.:60.00
#>  Max.   :80.00
```

Many other packages include their own version of descriptive statistics; one of my favorite is the `describe()` function from `psych`:

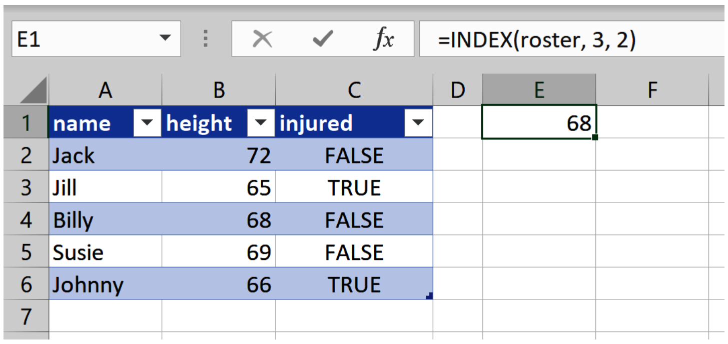
```
describe(star)
#>           vars    n   mean    sd median trimmed   mad min max range  skew
#> tmathssk     1 5748 485.65 47.77    484  483.20 44.48 320 626   306  0.47
#> treadssk     2 5748 436.74 31.77    433  433.80 28.17 315 627   312  1.34
#> classk*      3 5748   1.95  0.80      2    1.94  1.48   1   3     2  0.08
#> totexpk      4 5748   9.31  5.77      9    9.00  5.93   0  27    27  0.42
#> sex*         5 5748   1.49  0.50      1    1.48  0.00   1   2     1  0.06
#> freelunk*    6 5748   1.48  0.50      1    1.48  0.00   1   2     1  0.07
#> race*        7 5748   2.35  0.93      3    2.44  0.00   1   3     2 -0.75
#> schidkn      8 5748  39.84 22.96     39   39.76 29.65   1  80    79  0.04
#>           kurtosis   se
#> tmathssk      0.29 0.63
#> treadssk      3.83 0.42
#> classk*      -1.45 0.01
#> totexpk      -0.21 0.08
#> sex*         -2.00 0.01
#> freelunk*    -2.00 0.01
#> race*        -1.43 0.01
#> schidkn      -1.23 0.30
```

If you're not familiar with all of these descriptive statistics, you know what to do: *check the function's documentation*.

Indexing and Subsetting Data Frames

Earlier in this section we created a small data frame `roster` containing the names and heights of four individuals. Let's demonstrate some basic data frame manipulation techniques with this object.

In Excel, you can use the `INDEX()` function to refer to both the row and column positions of a table, as shown in Figure 7-4:


E1 =INDEX(roster, 3, 2)

	A	B	C	D	E	F
1	name	height	injured		68	
2	Jack	72	FALSE			
3	Jill	65	TRUE			
4	Billy	68	FALSE			
5	Susie	69	FALSE			
6	Johnny	66	TRUE			
7						

Figure 7-4. The `INDEX()` function on an Excel table

This will work similarly in R. We'll use the same bracket notation as we to with index vectors, but this time we'll refer to both the row and column position:

```
# Third row, second column of data frame
roster[3, 2]
#> [1] 68
```

Again, we can use the `:` operator to retrieve all elements within a given range:

```
# Second through fourth rows, first through third columns
roster[2:4, 1:3]
#>    name height injured
#> 2  Jill     65    TRUE
#> 3 Billy     68   FALSE
#> 4 Susie     69   FALSE
```

It's also possible to select an entire row or column by leaving its index blank, or to use the `c()` function to subset nonconsecutive elements:

```
# Second and third rows only
roster[2:3,]
#>    name height injured
#> 2  Jill     65    TRUE
#> 3 Billy     68   FALSE

# First and third columns only
roster[, c(1,3)]
#>     name injured
#> 1   Jack   FALSE
#> 2   Jill    TRUE
#> 3  Billy   FALSE
#> 4  Susie   FALSE
#> 5 Johnny    TRUE
```

If we just want to access one column of the data frame, we can use the `$` operator. Interestingly, this results in a *vector*:

```
roster$height
#> [1] 72 65 68 69 66
is.vector(roster$height)
#> [1] TRUE
```

This confirms that a data frame is indeed a list of vectors of equal length.

Other Data Structures in R

We've focused on R's vector and data frame structures as they are equivalents to Excel's ranges and tables and those you're most likely to work with for data analysis. There are, however, several other data structures in base R such as matrices and lists. To learn more about these structures and how they relate to vectors and data frames, check out Hadley Wickham's *Advanced R*, 2nd edition (Chapman & Hall).

Writing Data Frames

As mentioned earlier, it's typical to read data into R, operate on it, and then export the results elsewhere. To write a data frame to a *.csv* file, you can use the `write_csv()` function from `readr`:

```
# Write roster data frame to csv
write_csv(roster, 'output/roster-output-r.csv')
```

If you have the working directory set to the book's companion repository, you should find this file waiting for you in the *output* folder.

Unfortunately, the `readxl` package does not include a function to write data to an Excel workbook. We can, however, use `writexl` and its `write_xlsx()` function:

```
# Write roster data frame to csv
write_xlsx(roster, 'output/roster-output-r.xlsx')
```

Conclusion

In this chapter, you progressed from single-element objects, to larger vectors, and finally to data frames. While we'll be working with data frames for the remainder of the book, it's helpful to keep in mind that they are collections of vectors and behave largely in the same way. Coming up, you will learn how to analyze, visualize, and ultimately test relationships in R data frames.

Exercises

Do the following exercises to test your knowledge of data structures in R:

1. Create a character vector of five elements, and then access the first and fourth elements of this vector.
2. Create two vectors `x` and `y` of length 4, one containing numeric and the other logical values. Multiply them and pass the result to `z`. What is the result?
3. Download the `nycflights13` package from CRAN. How many datasets are included with this package?
    - One of these datasets is called `airports`. Print the first few rows of this data frame as well as the descriptive statistics.
    - Another is called `weather`. Find the 10th through 12th rows and the 4th through 7th columns of this data frame. Write the results to a *.csv* file and an Excel workbook.

CHAPTER 8

Data Manipulation and Visualization in R

American statistician Ronald Thisted once quipped: “Raw data, like raw potatoes, usually require cleaning before use.” Data manipulation takes time, and you’ve felt the pain if you’ve ever done the following:

- Select, drop, or create calculated columns
- Sort or filter rows
- Group by and summarize categories
- Join multiple datasets by a common field

Chances are, you’ve done all of these in Excel… *a lot*, and you’ve probably dug into celebrated features like `VLOOKUP()` and PivotTables to accomplish them. In this chapter, you’ll learn the R equivalents of these techniques, particularly with the help of `dplyr`.

Data manipulation often goes hand in hand with visualization: as mentioned, humans are remarkably adept at visually processing information, so it’s a great way to size up a dataset. You’ll learn how to visualize data using the gorgeous `ggplot2` package, which like `dplyr` is part of the `tidyverse`. This will put you on solid footing to explore and test relationships in data using R, which will be covered in Chapter 9. Let’s get started by calling in the relevant packages. We’ll also be using the *star* dataset from the book’s companion repository (*https://oreil.ly/lmZb7*) in this chapter, so we can import it now:

```
library(tidyverse)
library(readxl)

star <- read_excel('datasets/star/star.xlsx')
head(star)
#> # A tibble: 6 x 8
#>   tmathssk treadssk classk           totexpk sex   freelunk race  schidkn
#>      <dbl>    <dbl> <chr>              <dbl> <chr> <chr>    <chr>   <dbl>
#> 1      473      447 small.class            7 girl  no       white      63
#> 2      536      450 small.class           21 girl  no       black      20
#> 3      463      439 regular.with.aide      0 boy   yes      black      19
#> 4      559      448 regular               16 boy   no       white      69
#> 5      489      447 small.class            5 boy   yes      white      79
#> 6      454      431 regular                8 boy   yes      white       5
```

Data Manipulation with dplyr

`dplyr` is a popular package built to manipulate tabular data structures. Its many functions, or *verbs*, work similarly and can be easily used together. Table 8-1 lists some common `dplyr` functions and their uses; this chapter covers each of these.

Table 8-1. Frequently used verbs of `dplyr`

Function	What it does
`select()`	Selects given columns
`mutate()`	Creates new columns based on existing columns
`rename()`	Renames given columns
`arrange()`	Reorders rows given criteria
`filter()`	Selects rows given criteria
`group_by()`	Groups rows by given columns
`summarize()`	Aggregates values for each group
`left_join()`	Joins matching records from Table B to Table A; result is NA if no match found in Table B

For the sake of brevity, I won't cover all of the functions of `dplyr` or even all the ways to use the functions that we do cover. To learn more about the package, check out *R for Data Science* by Hadley Wickham and Garrett Grolemund (O'Reilly). You can also access a helpful cheat sheet summarizing how the many functions of `dplyr` work together by navigating in RStudio to Help → Cheatsheets → Data Transformation with `dplyr`.

Column-Wise Operations

Selecting and dropping columns in Excel often requires hiding or deleting them. This can be difficult to audit or reproduce, because hidden columns are easily overlooked, and deleted columns aren't easily recovered. The `select()` function can be used to choose given columns from a data frame in R. For `select()`, as with each of these functions, the first argument will be which data frame to work with. Additional arguments are then provided to manipulate the data in that data frame. For example, we can select *tmathssk*, *treadssk*, and *schidkin* from `star` like this:

```
select(star, tmathssk, treadssk, schidkn)
#> # A tibble: 5,748 x 3
#>    tmathssk treadssk schidkn
#>       <dbl>    <dbl>   <dbl>
#>  1      473      447      63
#>  2      536      450      20
#>  3      463      439      19
#>  4      559      448      69
#>  5      489      447      79
#>  6      454      431       5
#>  7      423      395      16
#>  8      500      451      56
#>  9      439      478      11
#> 10      528      455      66
#> # ... with 5,738 more rows
```

We can also use the `-` operator with `select()` to *drop* given columns:

```
select(star, -tmathssk, -treadssk, -schidkn)
#> # A tibble: 5,748 x 5
#>    classk            totexpk sex   freelunk race
#>    <chr>               <dbl> <chr> <chr>    <chr>
#>  1 small.class             7 girl  no       white
#>  2 small.class            21 girl  no       black
#>  3 regular.with.aide       0 boy   yes      black
#>  4 regular                16 boy   no       white
#>  5 small.class             5 boy   yes      white
#>  6 regular                 8 boy   yes      white
#>  7 regular.with.aide      17 girl  yes      black
#>  8 regular                 3 girl  no       white
#>  9 small.class            11 girl  no       black
#> 10 small.class            10 girl  no       white
```

A more elegant alternative here is to pass all unwanted columns into a vector, *then* drop it:

```
select(star, -c(tmathssk, treadssk, schidkn))
#> # A tibble: 5,748 x 5
#>    classk            totexpk sex   freelunk race
#>    <chr>               <dbl> <chr> <chr>    <chr>
#>  1 small.class             7 girl  no       white
#>  2 small.class            21 girl  no       black
#>  3 regular.with.aide       0 boy   yes      black
#>  4 regular                16 boy   no       white
#>  5 small.class             5 boy   yes      white
#>  6 regular                 8 boy   yes      white
#>  7 regular.with.aide      17 girl  yes      black
#>  8 regular                 3 girl  no       white
#>  9 small.class            11 girl  no       black
#> 10 small.class            10 girl  no       white
```

Keep in mind that in the previous examples, we've just been calling functions: we didn't actually assign the output to an object.

One more bit of shorthand for `select()` is to use the `:` operator to select everything between two columns, inclusive. This time, I will assign the results of selecting everything from *tmathssk* to *totexpk* back to `star`:

```
star <- select(star, tmathssk:totexpk)
head(star)
#> # A tibble: 6 x 4
#>   tmathssk treadssk classk            totexpk
#>      <dbl>    <dbl> <chr>               <dbl>
#> 1      473      447 small.class             7
#> 2      536      450 small.class            21
#> 3      463      439 regular.with.aide       0
#> 4      559      448 regular                16
#> 5      489      447 small.class             5
#> 6      454      431 regular                 8
```

You've likely created calculated columns in Excel; `mutate()` will do the same in R. Let's create a column *new_column* of combined reading and math scores. With `mutate()`, we'll provide the name of the new column *first*, then an equal sign, and finally the calculation to use. We can refer to other columns as part of the formula:

```
star <- mutate(star, new_column = tmathssk + treadssk)
head(star)
#> # A tibble: 6 x 5
#>   tmathssk treadssk classk            totexpk new_column
#>      <dbl>    <dbl> <chr>               <dbl>      <dbl>
#> 1      473      447 small.class             7        920
#> 2      536      450 small.class            21        986
#> 3      463      439 regular.with.aide       0        902
#> 4      559      448 regular                16       1007
#> 5      489      447 small.class             5        936
#> 6      454      431 regular                 8        885
```

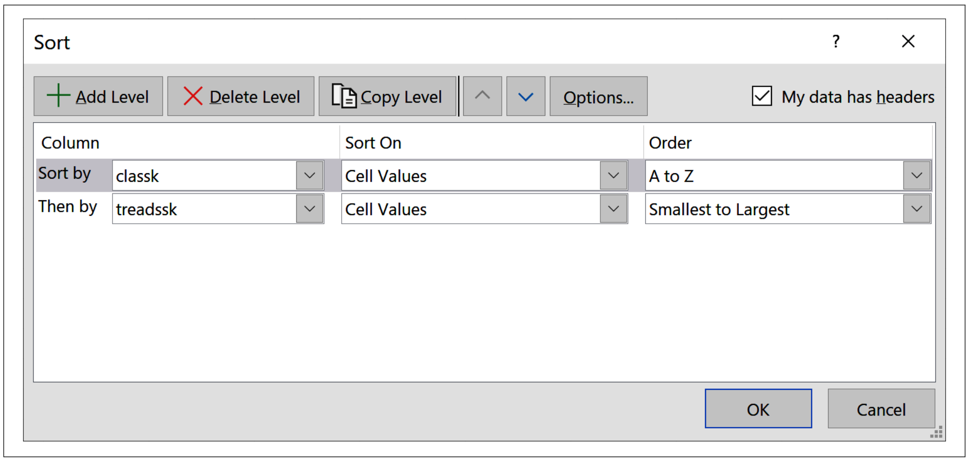
`mutate()` makes it easy to derive relatively more complex calculated columns such as logarithmic transformations or lagged variables; check out the help documentation for more.

new_column isn't a particularly helpful name for total score. Fortunately, the `rename()` function does what it sounds like it would. We'll specify what to name the new column in place of the old:

```
star <- rename(star, ttl_score = new_column)
head(star)
#> # A tibble: 6 x 5
#>   tmathssk treadssk classk            totexpk ttl_score
#>      <dbl>    <dbl> <chr>               <dbl>     <dbl>
#> 1      473      447 small.class             7       920
#> 2      536      450 small.class            21       986
#> 3      463      439 regular.with.aide       0       902
#> 4      559      448 regular                16      1007
#> 5       489       447 small.class            5        936
#> 6       454       431 regular                8        885
```

Row-Wise Operations

Thus far we've been operating on *columns*. Now let's focus on *rows*; specifically sorting and filtering. In Excel, we can sort by multiple columns with the Custom Sort menu. Say for example we wanted to sort this data frame by *classk*, then *treadssk*, both ascending. Our menu in Excel to do this would look like Figure 8-1.



Column		Sort On	Order
Sort by	classk	Cell Values	A to Z
Then by	treadssk	Cell Values	Smallest to Largest

Figure 8-1. The Custom Sort menu in Excel

We can replicate this in `dplyr` by using the `arrange()` function, including each column in the order in which we want the data frame sorted:

```
arrange(star, classk, treadssk)
#> # A tibble: 5,748 x 5
#>    tmathssk treadssk classk  totexpk ttl_score
#>       <dbl>    <dbl> <chr>     <dbl>     <dbl>
#>  1      320      315 regular       3       635
#>  2      365      346 regular       0       711
#>  3      384      358 regular      20       742
#>  4      384      358 regular       3       742
#>  5      320      360 regular       6       680
#>  6      423      376 regular      13       799
#>  7      418      378 regular      13       796
#>  8      392      378 regular      13       770
#>  9      392      378 regular       3       770
#> 10      399      380 regular       6       779
#> # ... with 5,738 more rows
```

We can pass the `desc()` function to a column if we'd like that column to be sorted descendingly.

```
# Sort by classk descending, treadssk ascending
arrange(star, desc(classk), treadssk)
#> # A tibble: 5,748 x 5
#>    tmathssk treadssk classk      totexpk ttl_score
#>       <dbl>    <dbl> <chr>         <dbl>     <dbl>
#>  1      412      370 small.class      15       782
#>  2      434      376 small.class      11       810
#>  3      423      378 small.class       6       801
#>  4      405      378 small.class       8       783
#>  5      384      380 small.class      19       764
#>  6      405      380 small.class      15       785
#>  7      439      382 small.class       8       821
#>  8      384      384 small.class      10       768
#>  9      405      384 small.class       8       789
#> 10      423      384 small.class      21       807
```

Excel tables include helpful drop-down menus to filter any column by given conditions. To filter a data frame in R, we'll use the aptly named `filter()` function. Let's filter `star` to keep only the records where `classk` is equal to `small.class`. Remember that because we are checking for equality rather than assigning an object, we'll have to use == and not = here:

```
filter(star, classk == 'small.class')
#> # A tibble: 1,733 x 5
#>    tmathssk treadssk classk      totexpk ttl_score
#>       <dbl>    <dbl> <chr>         <dbl>     <dbl>
#>  1      473      447 small.class       7       920
#>  2      536      450 small.class      21       986
#>  3      489      447 small.class       5       936
#>  4      439      478 small.class      11       917
#>  5      528      455 small.class      10       983
#>  6      559      474 small.class       0      1033
#>  7      494      424 small.class       6       918
#>  8      478      422 small.class       8       900
#>  9      602      456 small.class      14      1058
#> 10      439      418 small.class       8       857
#> # ... with 1,723 more rows
```

We can see from the tibble output that our `filter()` operation *only* affected the number of rows, *not* the columns. Now we'll find the records where `treadssk` is at least `500`:

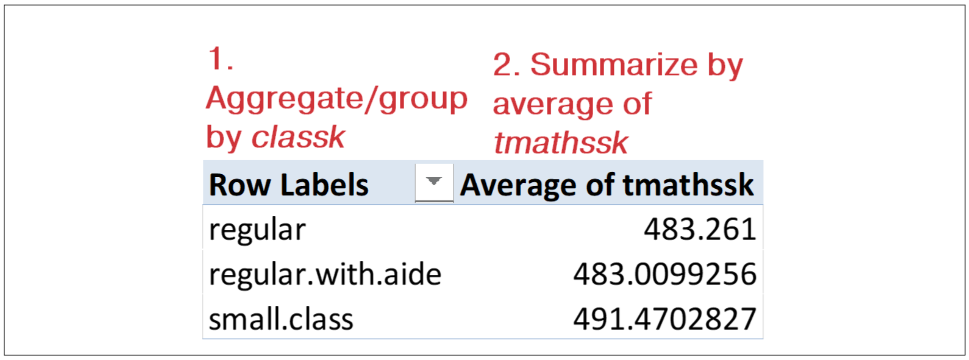
```
filter(star, treadssk >= 500)
#> # A tibble: 233 x 5
#>    tmathssk treadssk classk            totexpk ttl_score
#>       <dbl>    <dbl> <chr>               <dbl>     <dbl>
#>  1      559      522 regular                 8      1081
#>  2      536      507 regular.with.aide       3      1043
#>  3      547      565 regular.with.aide       9      1112
#>  4      513      503 small.class             7      1016
#>  5      559      605 regular.with.aide       5      1164
#>  6      559      554 regular                14      1113
#>  7      559      503 regular               10      1062
#>  8      602      518 regular               12      1120
#>  9      536      580 small.class           12      1116
#> 10      626      510 small.class           14      1136
#> # ... with 223 more rows
```

It's possible to filter by multiple conditions using the `&` operator for "and" along with the `|` operator for "or." Let's combine our two criteria from before with `&`:

```
# Get records where classk is small.class and
# treadssk is at least 500
filter(star, classk == 'small.class' & treadssk >= 500)
#> # A tibble: 84 x 5
#>    tmathssk treadssk classk      totexpk ttl_score
#>       <dbl>    <dbl> <chr>         <dbl>     <dbl>
#>  1      513      503 small.class       7      1016
#>  2      536      580 small.class      12      1116
#>  3      626      510 small.class      14      1136
#>  4      602      518 small.class       3      1120
#>  5      626      565 small.class      14      1191
#>  6      602      503 small.class      14      1105
#>  7      626      538 small.class      13      1164
#>  8      500      580 small.class       8      1080
#>  9      489      565 small.class      19      1054
#> 10      576      545 small.class      19      1121
#> # ... with 74 more rows
```

Aggregating and Joining Data

I like to call PivotTables "the WD-40 of Excel" because they allow us to get our data "spinning" in different directions for easy analysis. For example, let's recreate the PivotTable in Figure 8-2 showing the average math score by class size from the *star* dataset:


1. Aggregate/group by *classk*
2. Summarize by average of *tmathssk*

Row Labels	Average of tmathssk
regular	483.261
regular.with.aide	483.0099256
small.class	491.4702827

Figure 8-2. How Excel PivotTables work

As Figure 8-2 calls out, there are two elements to this PivotTable. First, I aggregated our data by the variable *classk*. Then, I summarized it by taking an average of *tmathssk*. In R, these are discrete steps, using different `dplyr` functions. First, we'll aggregate the data using `group_by()`. Our output includes a line, `# Groups: classk [3]`, indicating that `star_grouped` is split into three groups with the `classk` variable:

```
star_grouped <- group_by(star, classk)
head(star_grouped)
#> # A tibble: 6 x 5
#> # Groups:   classk [3]
#>   tmathssk treadssk classk            totexpk ttl_score
#>      <dbl>    <dbl> <chr>               <dbl>     <dbl>
#> 1      473      447 small.class             7       920
#> 2      536      450 small.class            21       986
#> 3      463      439 regular.with.aide       0       902
#> 4      559      448 regular                16      1007
#> 5      489      447 small.class             5       936
#> 6      454      431 regular                 8       885
```

We've *grouped* our data by one variable; now let's *summarize* it by another with the `summarize()` function (`summarise()` also works). Here we'll specify what to name the resulting column, and how to calculate it. Table 8-2 lists some common aggregation functions.

Table 8-2. Helpful aggregation functions for `dplyr`

Function	Aggregation type
`sum()`	Sum
`n()`	Count values
`mean()`	Average
`max()`	Highest value
`min()`	Lowest value
`sd()`	Standard deviation

We can get the average math score by class size by running `summarize()` on our grouped data frame:

```
summarize(star_grouped, avg_math = mean(tmathssk))
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 3 x 2
#>   classk            avg_math
#>   <chr>                <dbl>
#> 1 regular               483.
#> 2 regular.with.aide     483.
#> 3 small.class           491.
```

The `` `summarise()` ungrouping output`` error is a warning that you've ungrouped the grouped tibble by aggregating it. Minus some formatting differences, we have the same results as Figure 8-2.

If PivotTables are the WD-40 of Excel, then `VLOOKUP()` is the duct tape, allowing us to easily combine data from multiple sources. In our original *star* dataset, *schidkin* is a school district indicator. We dropped this column earlier in this chapter, so let's read it in again. But what if in addition to the indicator number we actually wanted to know the *names* of these districts? Fortunately, *districts.csv* in the book repository has this information, so let's read both in and come up with a strategy for combining them:

```
star <- read_excel('datasets/star/star.xlsx')
head(star)
#> # A tibble: 6 x 8
#>   tmathssk treadssk classk            totexpk sex   freelunk race  schidkn
#>      <dbl>    <dbl> <chr>               <dbl> <chr> <chr>    <chr>   <dbl>
#> 1      473      447 small.class             7 girl  no       white      63
#> 2      536      450 small.class            21 girl  no       black      20
#> 3      463      439 regular.with.aide       0 boy   yes      black      19
#> 4      559      448 regular                16 boy   no       white      69
#> 5      489      447 small.class             5 boy   yes      white      79
#> 6      454      431 regular                 8 boy   yes      white       5

districts <- read_csv('datasets/star/districts.csv')

#> -- Column specification -----------------------------------------------------
#> cols(
#>   schidkn = col_double(),
#>   school_name = col_character(),
#>   county = col_character()
#> )

head(districts)
#> # A tibble: 6 x 3
#>   schidkn school_name     county
#>     <dbl> <chr>           <chr>
#> 1       1 Rosalia         New Liberty
#> 2       2 Montgomeryville Topton
#> 3       3 Davy            Wahpeton
#> 4       4 Steelton        Palestine
#> 5       6 Tolchester      Sattley
#> 6       7 Cahokia         Sattley
```

It appears that what's needed is like a `VLOOKUP()`: we want to "read in" the *school_name* (and possibly the *county*) variables from *districts* into *star*, given the shared *schidkn* variable. To do this in R, we'll use the methodology of *joins*, which comes from relational databases, a topic that was touched on in Chapter 5. Closest to a `VLOOKUP()` is the left outer join, which can be done in `dplyr` with the `left_join()` function. We'll provide the "base" table first (*star*) and then the "lookup" table (*districts*). The function will look for and return a match in *districts* for every record in *star*, or return `NA` if no match is found. I will keep only some columns from *star* for less overwhelming console output:

```
# Left outer join star on districts
left_join(select(star, schidkn, tmathssk, treadssk), districts)
#> Joining, by = "schidkn"
#> # A tibble: 5,748 x 5
#>    schidkn tmathssk treadssk school_name     county
#>      <dbl>    <dbl>    <dbl> <chr>           <chr>
#>  1      63      473      447 Ridgeville      New Liberty
#>  2      20      536      450 South Heights   Selmont
#>  3      19      463      439 Bunnlevel       Sattley
#>  4      69      559      448 Hokah           Gallipolis
#>  5      79      489      447 Lake Mathews    Sugar Mountain
#>  6       5      454      431 NA              NA
#>  7      16      423      395 Calimesa        Selmont
#>  8      56      500      451 Lincoln Heights Topton
#>  9      11      439      478 Moose Lake      Imbery
#> 10      66      528      455 Siglerville     Summit Hill
#> # ... with 5,738 more rows
```

`left_join()` is pretty smart: it knew to join on `schidkn`, and it "looked up" not just *school_name* but also *county*. To learn more about joining data, check out the help documentation.

In R, missing observations are represented as the special value `NA`. For example, it appears that no match was found for the name of district 5. In a `VLOOKUP()`, this would result in an `#N/A` error. An `NA` does *not* mean that an observation is equal to zero, only that its value is missing. You may see other special values such as `NaN` or `NULL` while programming R; to learn more about them, launch the help documentation.

dplyr and the Power of the Pipe (%>%)

As you're beginning to see, `dplyr` functions are powerful and rather intuitive to anyone who's worked with data, including in Excel. And as anyone who's worked with data knows, it's rare to prepare the data as needed in just one step. Take, for example, a typical data analysis task that you might want to do with *star*:

> Find the average reading score by class type, sorted high to low.

Knowing what we do about working with data, we can break this into three distinct steps:

1. Group our data by class type.
2. Find the average reading score for each group.
3. Sort these results from high to low.

We could carry this out in `dplyr` doing something like the following:

```
star_grouped <- group_by(star, classk)
star_avg_reading <- summarize(star_grouped, avg_reading = mean(treadssk))
#> `summarise()` ungrouping output (override with `.groups` argument)
#>
star_avg_reading_sorted <- arrange(star_avg_reading, desc(avg_reading))
star_avg_reading_sorted
#>
#> # A tibble: 3 x 2
#>   classk           avg_reading
#>   <chr>                  <dbl>
#> 1 small.class             441.
#> 2 regular.with.aide       435.
#> 3 regular                 435.
```

This gets us to an answer, but it took quite a few steps, and it can be hard to follow along with the various functions and object names. The alternative is to link these functions together with the `%>%`, or pipe, operator. This allows us to pass the output of one function directly into the input of another, so we're able to avoid continuously renaming our inputs and outputs. The default keyboard shortcut for this operator is Ctrl+Shift+M for Windows, Cmd-Shift-M for Mac.

Let's re-create the previous steps, this time with the pipe operator. We'll place each function on its own line, combining them with `%>%`. While it's not necessary to place each step on its own line, it's often preferred for legibility. When using the pipe operator, it's also not necessary to highlight the entire code block to run it; simply place your cursor anywhere in the following selection and execute:

```
 star %>%
   group_by(classk) %>%
   summarise(avg_reading = mean(treadssk)) %>%
   arrange(desc(avg_reading))
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 3 x 2
#>   classk           avg_reading
#>   <chr>                  <dbl>
#> 1 small.class             441.
#> 2 regular.with.aide       435.
#> 3 regular                 435.
```

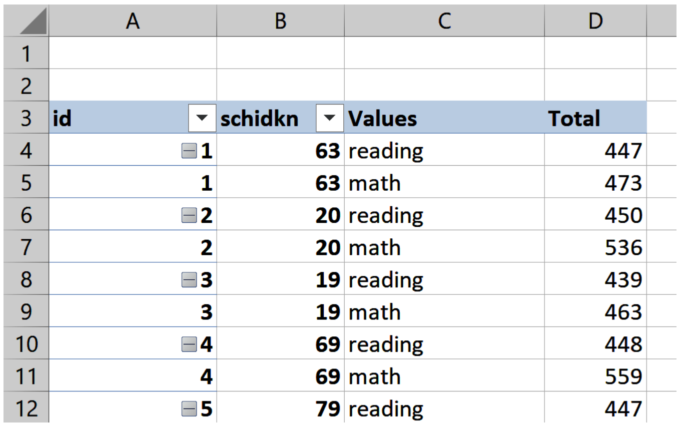
It can be pretty disorienting at first to no longer be explicitly including the data source as an argument in each function. But compare the last code block to the one before and you can see how much more efficient this approach can be. What's more, the pipe operator can be used with non-`dplyr` functions. For example, let's just assign the first few rows of the resulting operation by including `head()` at the end of the pipe:

```
# Average math and reading score
# for each school district
star %>%
   group_by(schidkn) %>%
    summarise(avg_read = mean(treadssk), avg_math = mean(tmathssk)) %>%
    arrange(schidkn) %>%
    head()
#> `summarise()` ungrouping output (override with `.groups` argument)
#> # A tibble: 6 x 3
#>   schidkn avg_read avg_math
#>     <dbl>    <dbl>    <dbl>
#> 1       1     444.     492.
#> 2       2     407.     451.
#> 3       3     441      491.
#> 4       4     422.     468.
#> 5       5     428.     460.
#> 6       6     428.     470.
```

Reshaping Data with tidyr

Although it's true that `group_by()` along with `summarize()` serve as a PivotTable equivalent in R, these functions can't do everything that an Excel PivotTable can do. What if, instead of just aggregating the data, you wanted to *reshape* it, or change how rows and columns are set up? For example, our *star* data frame has two separate columns for math and reading scores, *tmathssk* and *treadssk*, respectively. I would like to combine these into one column called *score*, with another called *test_type* indicating whether each observation is for math or reading. I'd also like to keep the school indicator, *schidkn*, as part of the analysis.

Figure 8-3 shows what this might look like in Excel; note that I relabeled the Values fields from *tmathssk* and *treadssk* to *math* and *reading*, respectively. If you would like to inspect this PivotTable further, it is available in the book repository as *ch-8.xlsx* (*https://oreil.ly/Kq93s*). Here I am again making use of an index column; otherwise, the PivotTable would attempt to "roll up" all values by *schidkn*.



	A	B	C	D
1				
2				
3	id	schidkn	Values	Total
4	⊟1	63	reading	447
5	1	63	math	473
6	⊟2	20	reading	450
7	2	20	math	536
8	⊟3	19	reading	439
9	3	19	math	463
10	⊟4	69	reading	448
11	4	69	math	559
12	⊟5	79	reading	447

Figure 8-3. Reshaping star in Excel

We can use `tidyr`, a core `tidyverse` package, to reshape *star*. Adding an index column will also be helpful when reshaping in R, as it was in Excel. We can make one with the `row_number()` function:

```
star_pivot <- star %>%
                select(c(schidkn, treadssk, tmathssk)) %>%
                mutate(id = row_number())
```

To reshape the data frame, we'll use `pivot_longer()` and `pivot_wider()`, both from `tidyr`. Consider in your mind's eye and in Figure 8-3 what would happen to our dataset if we consolidated scores from *tmathssk* and *treadssk* into one column. Would the dataset get longer or wider? We're adding rows here, so our dataset will get longer. To use `pivot_longer()`, we'll specify with the `cols` argument what columns to lengthen by, and use `values_to` to name the resulting column. We'll also use `names_to` to name the column indicating whether each score is math or reading:

```
star_long <- star_pivot %>%
                 pivot_longer(cols = c(tmathssk, treadssk),
                              values_to = 'score', names_to = 'test_type')
head(star_long)
#> # A tibble: 6 x 4
#>   schidkn    id test_type score
#>     <dbl> <int> <chr>     <dbl>
#> 1      63     1 tmathssk    473
#> 2      63     1 treadssk    447
#> 3      20     2 tmathssk    536
#> 4      20     2 treadssk    450
#> 5      19     3 tmathssk    463
#> 6      19     3 treadssk    439
```

Great work. But is there a way to rename *tmathssk* and *treadssk* to *math* and *reading*, respectively? There is, with `recode()`, yet another helpful `dplyr` function that can be used with `mutate()`. `recode()` works a little differently than other functions in the package because we include the name of the "old" values *before* the equals sign, then the new. The `distinct()` function from `dplyr` will confirm that all rows have been named either *math* or *reading*:

```
# Rename tmathssk and treadssk as math and reading
star_long <- star_long %>%
   mutate(test_type = recode(test_type,
                            'tmathssk' = 'math', 'treadssk' = 'reading'))

distinct(star_long, test_type)
#> # A tibble: 2 x 1
#>   test_type
#>   <chr>
#> 1 math
#> 2 reading
```

Now that our data frame is lengthened, we can widen it back with `pivot_wider()`. This time, I'll specify which column has values in its rows that should be columns with `values_from`, and what the resulting columns should be named with `names_from`:

```
star_wide <- star_long %>%
                pivot_wider(values_from = 'score', names_from = 'test_type')
head(star_wide)
#> # A tibble: 6 x 4
#>   schidkn    id  math reading
#>     <dbl> <int> <dbl>   <dbl>
#> 1      63     1   473     447
#> 2      20     2   536     450
#> 3      19     3   463     439
#> 4      69     4   559     448
#> 5      79     5   489     447
#> 6       5     6   454     431
```

Reshaping data is a relatively trickier operation in R, so when in doubt, ask yourself: *am I making this data wider or longer? How would I do it in a PivotTable?* If you can logically walk through what needs to happen to achieve the desired end state, coding it will be that much easier.

Data Visualization with ggplot2

There's so much more that `dplyr` can do to help us manipulate data, but for now let's turn our attention to data visualization. Specifically, we'll focus on another `tidyverse` package, `ggplot2`. Named and modeled after the "grammar of graphics" devised by computer scientist Leland Wilkinson, `ggplot2` provides an ordered approach for constructing plots. This structure is patterned after how elements of speech come together to make a sentence, hence the "grammar" of graphics.

I'll cover some of the basic elements and plot types of `ggplot2` here. For more about the package, check out *ggplot2: Elegant Graphics for Data Analysis* by the package's original author, Hadley Wickham (Springer). You can also access a helpful cheat sheet for working with the package by navigating in RStudio to Help → Cheatsheets → Data Visualization with ggplot2. Some essential elements of `ggplot2` are found in Table 8-3. Other elements are available; for more information, check out the resources mentioned earlier.

Table 8-3. The foundational elements of `ggplot2`

Element	Description
`data`	The source data
`aes`	The aesthetic mappings from data to visual properties (x- and y-axes, color, size, and so forth)
`geom`	The type of geometric object observed in the plot (lines, bars, dots, and so forth)

Let's get started by visualizing the number of observations for each level of *classk* as a barplot. We'll start with the `ggplot()` function and specify the three elements from Table 8-3:

```
ggplot(data = star, ❶
          aes(x = classk)) + ❷
   geom_bar() ❸
```

❶ The data source is specified with the `data` argument.

❷ The aesthetic mappings from the data to the visualization are specified with the `aes()` function. Here we are calling for *classk* to be mapped to the x-axis of the eventual plot.

❸ We plot a geometric object based on our specified data and aesthetic mappings with the `geom_bar()` function. The results are shown in Figure 8-4.

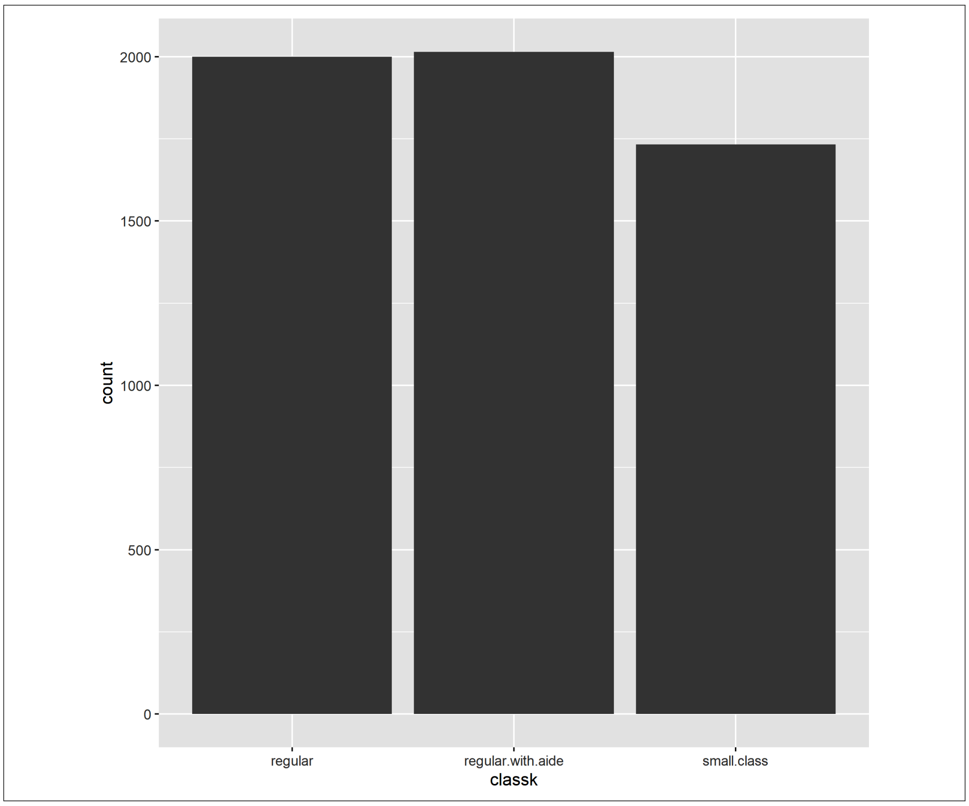


Figure 8-4. A barplot in `ggplot2`

Similar to the pipe operator, it's not necessary to place each layer of the plot on its own line, but it's often preferred for legibility. It's also possible to execute the entire plot by placing the cursor anywhere inside the code block and running.

Because of its modular approach, it's easy to iterate on visualizations with `ggplot2`. For example, we can switch our plot to a histogram of *treadssk* by changing our `x` mapping and plotting the results with `geom_histogram()`. This results in the histogram shown in Figure 8-5:

```
ggplot(data = star, aes(x = treadssk)) +
  geom_histogram()

#> `stat_bin()` using `bins = 30`. Pick better value with `binwidth`.
```

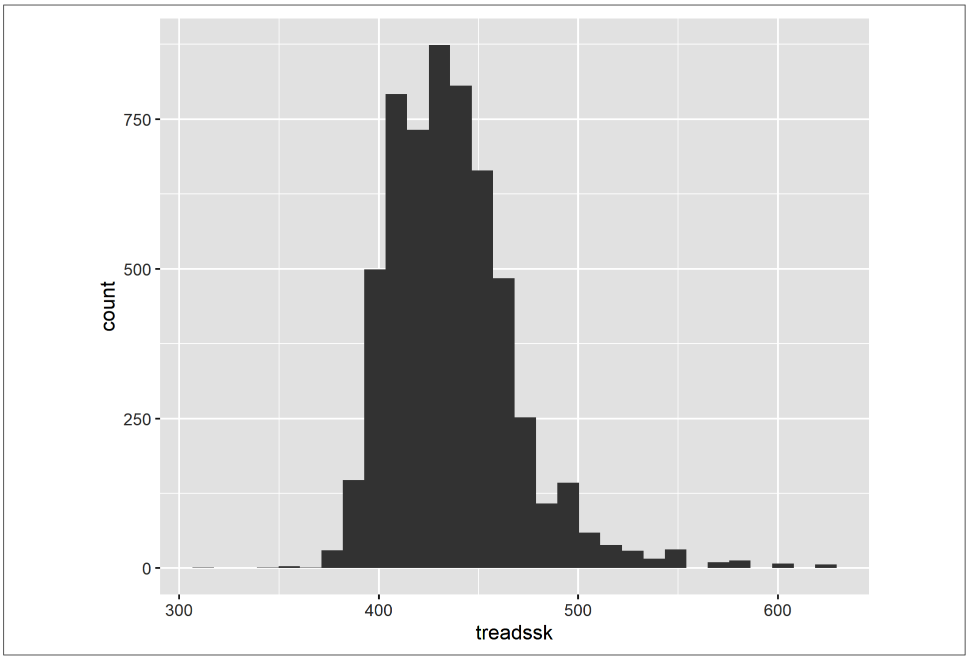


Figure 8-5. A histogram in `ggplot2`

There are also many ways to customize `ggplot2` plots. You may have noticed, for example, that the output message for the previous plot indicated that 30 bins were used in the histogram. Let's change that number to 25 and use a pink fill with a couple of additional arguments in `geom_histogram()`. This results in the histogram shown in Figure 8-6:

```
ggplot(data = star, aes(x = treadssk)) +
  geom_histogram(bins = 25, fill = 'pink')
```

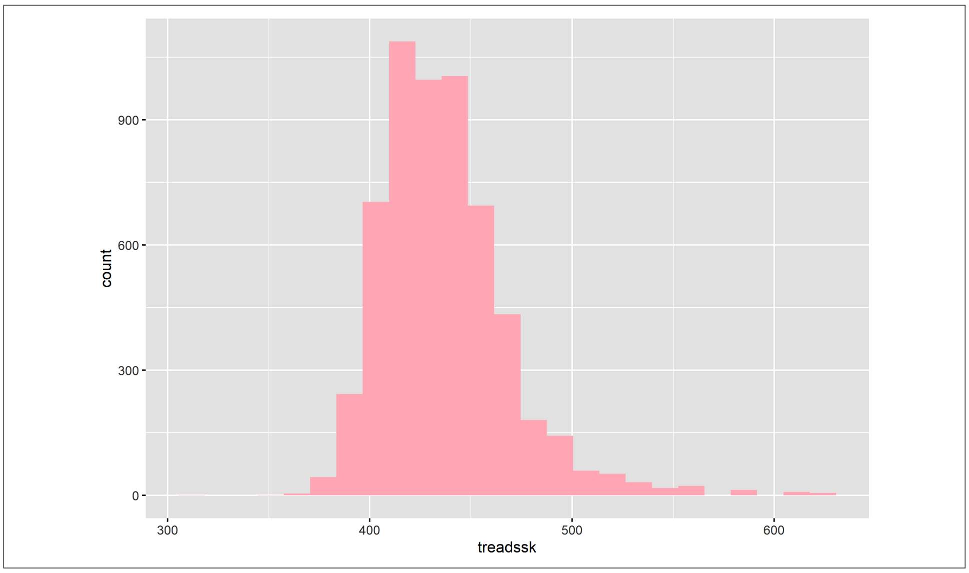


Figure 8-6. A customized histogram in `ggplot2`

Use `geom_boxplot()` to create a boxplot, as shown in Figure 8-7:

```
ggplot(data = star, aes(x = treadssk)) +
  geom_boxplot()
```

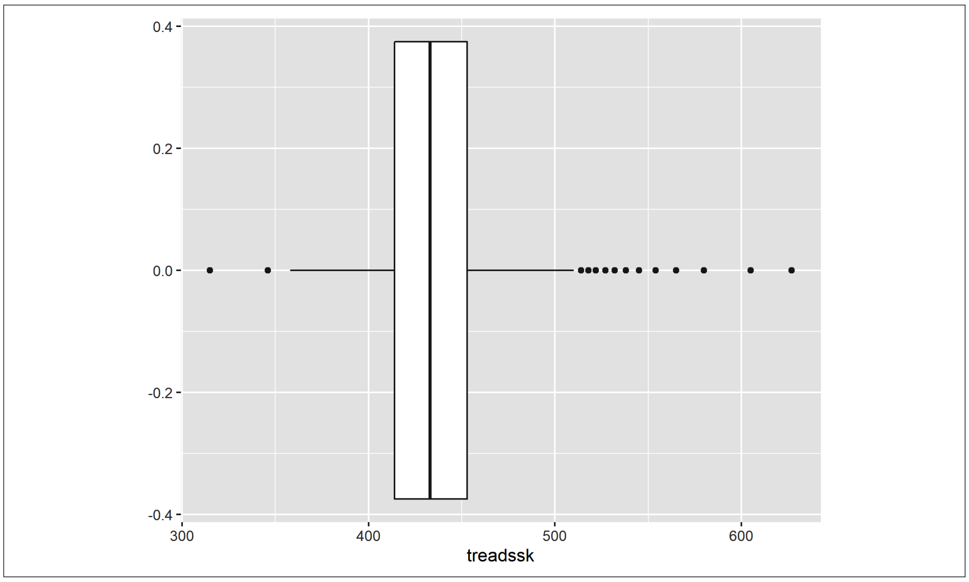


Figure 8-7. A boxplot

In any of the cases thus far, we could have "flipped" the plot by including the variable of interest in the `y` mapping instead of the `x`. Let's try it with our boxplot. Figure 8-8 shows the result of the following:

```
ggplot(data = star, aes(y = treadssk)) +
  geom_boxplot()
```

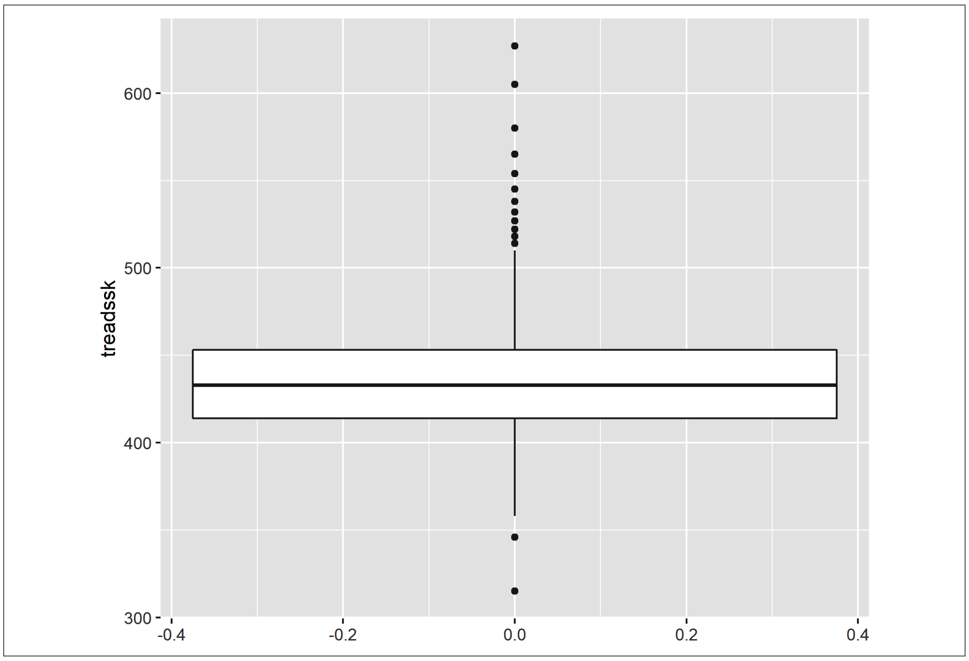


Figure 8-8. A "flipped" boxplot

Now let's make a boxplot for each level of class size by mapping *classk* to the x-axis and *treadssk* to the y, resulting in the boxplot shown in Figure 8-9:

```
ggplot(data = star, aes(x = classk, y = treadssk)) +
  geom_boxplot()
```

Similarly, we can use `geom_point()` to plot the relationship of *tmathssk* and *treadssk* on the x- and y-axes, respectively, as a scatterplot. This results in Figure 8-10:

```
ggplot(data = star, aes(x = tmathssk, y = treadssk)) +
  geom_point()
```