

---

# 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:

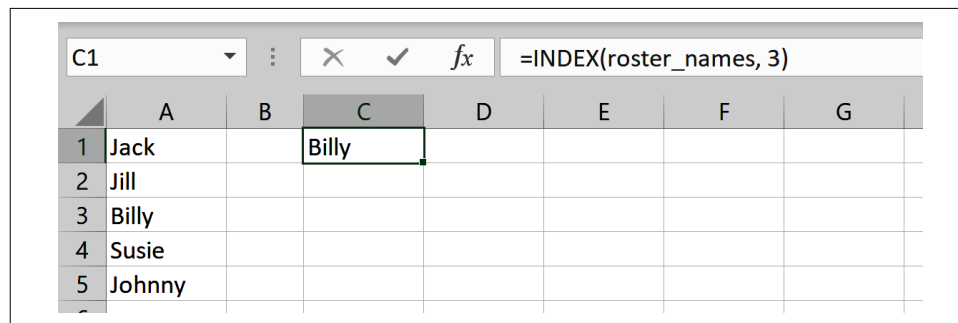
```
my_vec <- c('A', 2, 'C')
my_vec
#> [1] "A" "2" "C"

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):



	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 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 *.xls**m* 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
#> classsk*     -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:

	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.