1

R

1.1 Introduction

This book focuses on one problem that is common to almost every statistical problem – indeed, to almost any problem involving any sort of analysis. That problem is acquiring and preparing the data. Across our many years of data analysis, we have learned that seemingly 80% of our time – maybe more – goes into the data preparation steps (a belief echoed by others such as Dasu and Johnson, 2003). Collectively, we call these actions *data cleaning*, although, as we will discuss later, we sometimes use that term for something a little more specific. Regardless of the name, almost any analysis requires that you (i) acquire that data, that is, read it into the computer program; (ii) clean the data, that is, identify entries that are duplicated or clearly erroneous or anomalous, and take other preparation steps (e.g., combining entries such as "Female," "female," and "F"); (iii) merge data from different sources; and (iv) prepare the data for modeling, which might involve dividing a set of numeric values into subsets, combining states into regions, and so on. This book discusses some approaches for accomplishing these four steps in the R language (R Core Team, 2013). A fifth problem, which receives less emphasis, is the problem of long-term curation of the data. Which parts of the data must be saved and in what way? We address that question by reference to the idea of *reproducible research*, which we discuss later in this chapter, and later in the book as well.

1.1.1 What Is R?

R is a computer program that lets you analyze data. By "analyze" we mean, first, read the data into the program and then operate on it – drawing graphs and charts, manipulating values, fitting statistical models, and so on. (Notice that we prefer to call data "it" rather than "them." We discuss this choice briefly toward the end of the chapter.) R is both a statistical "environment" and also

A Data Scientist's Guide to Acquiring, Cleaning, and Managing Data in R, First Edition.
Samuel E. Buttrey and Lyn R. Whitaker.

Companion website: www.wiley.com/go/buttrey/datascientistsguide

a programming language, and it is very widely used both in commercial and academic settings. R is free and open-source and runs on Windows, Apple, and Linux operating systems. It is maintained by a group of volunteers who release bug fixes and new features regularly.

1.1.2 Who Uses R and Why?

R started as a tool for statisticians, evolving from a language called S that was created in the 1970s. Today, R remains the primary language of academic statisticians, and it also has a prominent place among analysts in business and government as well. It is used not only for building statistical models but also for handling and cleaning data, as in this book, and for developing new statistical methods, building simulations, for visualization, and generally for all the data-handling tools the statistician and the data scientist require. Because of the ease with which users can develop and distribute new methods, R has also become the tool of choice in certain fast-growing fields such as biostatistics and genetics. Articles on "surveys of the top tools used by data scientists" inevitably name R as one of the important tools with which data scientists, as well as statisticians, should be familiar. Moreover, R's popularity is such that there are extensions to R (see "packages" in Section 1.4.4) that allow you to connect to other programs such as the Python and Java languages, the H2O machine-learning system, the ArcGIS geographical information system, and many more.

1.1.3 Acquiring and Installing R

The primary way to acquire R is to download it from the Internet. The main R website for R is `www.r-project.org`, and the `www.cran.r-project.org` page ("CRAN" standing for "Comprehensive R Archive Network") is where you can download R itself. There are in fact dozens of "mirror" sites for CRAN – that is, websites that are essentially copies of the CRAN site – so as to reduce the load on the CRAN site. You can probably find a mirror near you on the "mirrors" page. After you download R, install it in the way you would normally install a program on your operating system.

At any one time, users around the world will be running slightly different versions of R, since new ones are released fairly frequently. For example, at this writing the current version of R was called 3.3.2, but many users are still using 3.2 or earlier versions. This will almost never cause problems, but it is a good idea to update your version of R from time to time.

There are also several slightly different versions of R distributed other than at CRAN. Microsoft R Open is a particular version of R that uses a different set of math libraries intended to make certain computations faster. Like "regular" R, Microsoft R Open is free, although it does not run on OS X. Other versions of R are intended to communicate with relational databases or with other

big-data platforms. For this book, we will assume you are running "regular" R – but in any case for our purposes all versions of R should behave exactly the same way.

1.1.4 Starting and Quitting R

The way you start R depends on your operating system. Normally double-clicking on an R icon will be enough to get R started. In the command-line interface of many Linux systems, or using the OS X terminal window, it may be enough just to type the upper-case letter `R` (or, for Windows command lines, `Rgui`). When R has started, you will see the command prompt `>`. This is the R *console*, the place where commands are entered. At this point, you can start typing commands to R. When it comes time to quit R, you can either "kill" the window in the usual way (for OS X, the red dot, the lightswitch in the top right, or via the File dialog; for Windows, the red X or File dialog) or you can type the `q()` command. In either case, R will then ask you if you want to "Save workspace image." If you answer "yes" to this question, R will save to the disk any changes you made during the current session, whereas if you answer "no," R will return its workspace to the condition it was in when R was last started. We almost always want to answer "yes" to this question!

1.2 Data

Data is information about the elements of whatever problem we are investigating. Data comes in many forms, but for our purposes it will always be presented in a set of computer-ready values. For example, a database concerning birds might include text about the habits of the birds, numbers giving lengths and weights of the individuals, maps showing migration patterns, images showing the birds themselves, sound recordings of the birds' calls, and so on. Although they look very different, all of these different pieces of information can be represented in the computer in digital form in one way or another. In this example, one of our primary tasks might be to ensure that each bird's description is correctly matched with the correct map, image, and song file. Our data analysis projects rarely include data quite so disparate, but in almost every case we need to acquire data, clean it (a process we start to describe in what follows and continue throughout the book), and prepare it for modeling, and in almost every case we expect our data to consist of both numeric and textual values.

1.2.1 Acquiring Data

The first step in a data analysis project, of course, is to get the data into R where it can be manipulated. We are old enough to remember the days when this involved typing all the data from the back of a book or journal paper into a

statistics package by hand, but happily this is not necessary today. On the other hand, data now comes in a variety of formats, few of which were created with the convenience of the data scientist in mind. In Chapter 6, we describe some of these common formats and how to use R to read data effectively.

1.2.2 Cleaning Data

We "clean" data when we detect (and, in many cases, remove) anomalies. Anomalies will very often be missing values, but they might also be absurd ones, as when people's ages are reported as 999 or −1. Sometimes, as in our earlier example, we might have genders reported as "Female," "female," and "F" and we want to combine these three values. In the cleaning process we might learn, for example, that one data source produced no data at all in August 2016; this sort of fact will need to be brought to the attention of the data provider. The data cleaning process also involves merging data from different sources, extracting subsets or reshaping the data in some way. All in all, data cleaning is the process of turning raw data, received from one or more providers, into a data set that can be used in visualization, modeling, and decision-making.

In practice these steps are iterative. Our cleaning process not only informs the modeling, but it sometimes leads us to re-acquire the data in a different, more usable form. Similarly, insights from modeling will often lead us to prepare the data in a new and more revealing way – because it is when we model that we often discover anomalies or other interesting attributes of the data.

1.2.3 The Goal of Data Cleaning

What a "clean" data set should look like depends on what your goals are. One useful perspective is given by Wickham (2014), who describes what he calls "tidy" data. A tidy data set is rectangular (or tabular); each row describes one unit of analysis (an observation), and each column gives one measurement (a variable). For example, in a data set giving measurements about people, each row would concern itself with a person, and the columns might give height, weight, age, blood type, and so on.

In some problems, it is not immediately clear what the unit of analysis is. For example, imagine data that describes the locations of boats over the course of a month, as recorded by GPS. For some purposes, a "tidy" data set would have one row per GPS ping, each row giving a ship identifier, a location, and a time. For other purposes, we might prefer a data set with one row per boat, each row giving the southernmost point that ship reaches, or perhaps giving a binary indicator of whether the ship did, or did not, spend time in international waters. Some data – images and sound, for example – do not lend themselves to this "tidy" approach.

The exact layout of your final data will depend on what you plan to do with it – and in some cases this won't be known until after you have operated on the data.

1.2.4 Making Your Work Reproducible

It is vital that other people be able to reproduce the actions you took on your data. Ideally, you or another analyst should be able to start with your raw data, run all the steps you applied to it, and emerge with exactly the same clean, prepared data sets. This will be useful to you when you encounter a situation similar to the one in the previous paragraph, where the form of the new data needs to be designed. But it is even more important for another analyst, since if you or another analyst can reproduce your results there will be no disagreement about the data. The act of making research reproducible has, in recent years, been rightfully recognized as a cornerstone of scientific progress. Record and document every step you take so that others can repeat them.

1.3 The Very Basics of R

This book is about handling data in R. It cannot teach you the very basics of R in detail – although, happily, there are many good books and online resources that can. (We give a few examples at the end of this chapter.) In this section, we list a few of the most basic facts about R, but, again, this book is not intended to teach you R. Rather, we focus on the details of R and of the way data is represented in R, in order to help you understand some of the ways to acquire, clean, and handle data inside R.

1.3.1 Top Ten Quick Facts You Need to Know about R

In this section, we give a few of the most important facts about R a beginner needs to know. There will be more detail on these facts later in the chapter and throughout the book.

1) The **prompt** is (by default) >. If you leave a command incomplete, maybe because there is an unclosed parenthesis or quotation mark, R gives you the continuation prompt, which is +. The Esc key (Windows) or control-C (other systems) produces the **break** command, which will take you back to the regular prompt. In this example, we show what a completed command looks like – in this case, R is computing the value of 3 divided by 2.

```
> 3 / 2
[1] 1.5
```

Here, R produced the prompt (>), and we typed `3 / 2` and pressed the Enter (or "Return") key. R then produced the output. We will talk about the `[1]` part in Chapter 2, but the computed value of `1.5` is shown. In the following example, we show what happens when we press Enter after typing the slash character:

```
> 3/
+ 2
[1] 1.5
```

Here, since the expression on the first line was incomplete, R produced the continuation prompt, +. When we typed `2` and hit Enter, the expression was complete and the result was shown. In case of confusion, press break until the original > prompt is showing.

In examples in this book where we want to show the R output, we also show the > prompt in front of our code. Remember, that > is produced by R; you don't need to type that yourself. (At the end of the chapter, we tell you where you can get all the code from the book in electronic form.)

2) R is **case-sensitive**, which means that upper- and lower-case letters are different in R. For example, the built-in R object `LETTERS` gives all 26 upper-case letters. A different item called `letters` contains the lower-case versions of the alphabet. There is no built-in object called `Letters`.
3) **Show** an object by typing its name. For example, if you type `ls` by itself, you see the contents of the function whose name is `ls`, the one that lists all the objects in your workspace (which we define later). To actually run the function and see the objects, you need to type the function's name together with parentheses. In this case, list your objects by typing `ls()`.
4) Get **help** for a function or object named `thing` with the command `help(thing)` or `?thing`. For example, to see the help for the `ls()` function, type `help(ls)`. If you don't know the name, try `help.search()` with a relevant word in quotation marks; for example, try `help.search ("matrices")` to see functions that handle matrices.
5) **Assign** a value or object to a name with the left-arrow (less-than plus hyphen): for example, the command `a <- 1` creates a new object named `a` with value 1. (You can also assign with a command such as `a = 1`, but we don't recommend it.) The assignment will over-write any existing object named `a` you might have had. Once you create an object, it is in your "workspace," and your workspace can be saved when you quit. So unless your computer crashes, when you create an object it will persist until you delete it. **Display** the set of objects in your workspace with `objects()` or `ls()`; remove an object with `remove()` or `rm()`. Not every character is permitted in the name of an R object. Start a name

with a letter or a dot, and then stick to numbers, letters, underscores, and dots. Names cannot contain spaces. In this example, we show some assignments that succeed and some that do not.

```
> a <- 1
> a.1 <- 1
> 2a <- 1
Error: unexpected symbol in "2a"
> a 2 <- 1
Error: unexpected numeric constant in "a 2"
```

The first two of these assignments succeed, because `a` and `a.1` are valid names. The last two fail because they refer to invalid names.

6) The **comment character** is #. A comment ends at the end of the line. If you want a comment to span multiple lines, you need to start each comment line with #.
7) **Recall** earlier commands with the up-arrow. You can edit an earlier command and then press the Enter key to run the new version. The `history()` command shows a list of your recent commands; put a number in (as in `history(500)`) to see more.
8) When referring to file names, R itself uses the **forward slash** in the console. The Windows file system uses the backward slash, so Windows users may use that, too, but in that case you have to type `\\` (we talk more about this later on). For example, a Windows user who wants to access a file named `c:\temp\mycode.R` in an R command will need to type either `c:/temp/mycode.R` or `c:\\temp\\mycode.R`. You'll need to use a regular, single backslash if you are interacting with the Windows operating system and not R – if, for example, you are presented with a graphical "select file" window. The file systems for OS X and Linux users use the forward slash at all times.
9) Just about any function you want is built into R, so R makes an excellent **calculator**. For example,

```
> sin (log (34))
[1] -0.375344
```

This says that the sine (using radians) of the logarithm (base *e*) of 34 is −0.375344. Most functions allow you to specify "arguments," values you pass to the function to modify its behavior. Some must be specified; others have default values. For example, `log (34, 10)` produces the base 10 logarithm instead of the natural logarithm. If a function accepts multiple arguments, you will need to specify them in the proper order – or by name. In this example, the arguments to `log` are named `x` and `base` (see the help at `?log`), so we could have entered `log(base = 10, x = 34)` too.

10) R's **operators** include the comparison operators `!=` for "not equal," == for "is equal to," <= and >= for "less than or equal to" and "greater than or equal to," and the arithmetic operators `*` for "multiplied by" and `^` for "raised to the power of."

1.3.2 Vocabulary

As we get started, it will be worthwhile for us to repeat some of the vocabulary of R, and of data, that you should be familiar with. In this section, we define some of the terms that are commonly used in discussion of R, both in this book and elsewhere.

vector A *vector* is the simplest piece of data in R. It consists of one or more entries (also called "items" or "elements") that are all either text or all numbers or all "logical" (i.e., `TRUE` or `FALSE`). (Technically, a vector might have length 0, and there are some other types, but that last sentence covers 99% of what you will do with R.) For example, the value of the famous constant π is built into R as the object `pi`, and the R object `pi` is a numeric vector with length 1. We talk about vectors in Chapter 2.

matrix A *matrix* is just a two-dimensional vector in rectangular shape. While matrices are important in statistics, they are less important in the data cleaning process. Still, it is useful to know about matrices in preparation for using data frames (below). We discuss matrices at the start of Chapter 3.

list A *list* is an R object that can hold other R objects. Lists are everywhere in R and you will need to know how to create them and access their elements. We discuss lists starting in Section 3.3.

data frame A *data frame* is a cross between a matrix and a list. Like a matrix, it is rectangular, but like a list it can contain items of different sorts – numeric, text, and so on – as its columns. You can think of a data frame as a list of vectors all of which are the same length. Most of the data we encounter will be in the form of data frames, and, if it isn't, we will usually try to put it into a data frame. We talk about data frames starting in Section 3.4.

object An *object* is a general word for anything in R. Usually, we will use this to refer to data objects such as vectors, matrices, lists, or data frames, but we might use "object" to refer to a function, a file handle, or anything else with a name in R.

rows and columns A data frame and a matrix are two-dimensional rectangular objects, consisting of *rows* and *columns*. Our goal, in a data cleaning problem, is almost always to produce one or more data frames whose rows correspond to the things being measured, and whose columns give the different measurements. For example, in a military manpower problem each row might represent a soldier, and the columns would give measurements such as age, sex, rank, and years in service. Statisticians sometimes call rows and columns "observations" and "variables" (although that second

word has another meaning in R, see the following discussion). Confusingly, other terms exist too: authors in machine learning talk of "instances" (or "entities") and "attributes" ("features"). We will use "rows" and "columns" when the emphasis is on the representation of the data in a data frame, and "observations" and "variables" when the emphasis is on the role being played by the data.

variable A *variable* is also a generic term for an R object, especially one of the objects in our workspace. The name is slightly misleading because the object's value doesn't have to change. We would call `pi` a "variable," at least in casual conversation.

operator An *operator* describes an action on one or two objects – often vectors – and produces a result. For example, the `*` operator, placed between two numbers, produces their product. Most operators act on two things – we say they are "binary." The `+` and `-` operators can also be "unary," meaning they act on one number. So in the expression `-3`, the `-` is a unary operator. Operations are often "vectorized," meaning they act separately on each item of a vector.

function A *function* is a kind of R object that can take an action. Functions often accept arguments to control the computations they make, and produce "return values," the results of the computation. For example, the `cos()` function takes as its one argument the size of an angle, in radians, and produces, as its return value, the cosine of that angle. So typing `cos(1)` invokes a function and produces a value of about `0.54`. Operators are functions, too, although they don't look like it. For example, you can multiply two numbers by calling the `*` function explicitly with two arguments, though you'll need quotation marks; `"*"(3, 4)` operates `*` on 3 and 4 and produces 12. Functions are covered in detail in Chapter 5.

expression An *expression* is a legal R "phrase" that would produce an action if you entered it into R. For example, `a <- 3` is an expression that, if evaluated, would cause an item `a` to be created and given the value 3. That expression is called an *assignment*. `pi > 3` is an expression that would produce TRUE, since the number `pi` is greater than 3. This is an example of a *comparison*. Just typing `2` is also an expression; the system interprets this as being the same as `print(2)`, and prints out the value 2. Most expressions involve the use of functions or operators, as well as R variables.

command We often use the word "command" as a casual shortcut to mean "function," "operator," or "expression." For example, we might say "use the help command" instead of "run the help function."

script A *script* is a text file that can list R commands. We use script files in all of our projects and we recommend that you do, too. We discuss scripts in Chapter 5.

workspace The *workspace* is the set of objects (data and functions) in our current environment. These are objects we have created.

working directory The *working directory* is the folder on your computer where your R data is stored. By default, R will look in this directory for any external files you might ask for. We talk more about the working directory in the following section.

With this vocabulary in mind it is easier to discuss some of the ways that R operates. As an example, it's not always obvious what the different operators in R will do in weird cases. We know that `3 < 10` is `TRUE`. What is the value of `3 < "10"`? The answer is `FALSE`. R cannot compare a number to a character, so converts both values into characters. Then the comparison is made alphabetically. So just as `"Apple" < "Banana"` is `TRUE` because `"Apple"` comes first in alphabetical order, so too does `"10"` come before `"3"` – since, as always, we compare the initial characters first, and the `1` character precedes the `3` character in our computer's sorting system. We talk much more about the different types of data in R, and converting between them, in Chapter 2.

Another example of unexpected behavior has to do with the way R reads commands typed in at the command line. We saw that the command `a <- 3` assigns the value `3` to an object `a`. However, what happens when you type `a < - 3`, with a space between `<` and `-`? The answer is that R attaches the hyphen to the value `3`, and then compares the value of `a` to the number `-3`. In general, spaces will not affect your R commands – but in this case the space "broke" the assignment operator `<-`.

R objects have names and names have to conform to a small set of rules. If data is brought in from outside R, perhaps from a spreadsheet, names will be changed if they need to be made valid (details can be seen in the help for the `make.names()` function). Technically it is possible to force R to use invalid names, but don't do that. A few names in R are *reserved*, meaning they cannot be used as the name of an R variable. For example, you cannot name an object `TRUE`; that name is reserved. (You may name an object `T`, because that name isn't reserved, but we don't recommend it.) It is also wise to try to avoid giving an object the name of an existing R function (although there are lots of R functions and some are obscure). If you name a vector `sum`, and then use the `sum()` function to add things up, R will be smart enough to differentiate your vector from the system's function. But if you create a function called `sum()` in your workspace, R will use that one (since your function will appear first on the search path; see "search path" in Section 1.4.1). This is almost never what you want. The R functions `c()` and `t()` provide good examples of names to avoid.

Finally, R can operate in an "object-oriented" way. A number of R functions are "generic," meaning that have specific methods to handle specific data types. For example, the `summary()` function applied to a numeric vector gives some information about the values in the vector, but the same function applied to the output of a modeling function will often give summary statistics about the model. The exact action that the generic function takes depends on the "class"

(i.e., the type) of the object passed to it. We run across a few of these generic functions in the following few chapters and discuss object-oriented programming briefly in Section 5.6.3

1.3.3 Calculating and Printing in R

R performs calculations and prints results. In this section, we talk about some of the differences between what R computes and what it prints, as well as how text data is represented.

Floating-Point Error

This is a good place to discuss an issue that arises in a lot of data cleaning problems and has caught us and our students off-guard more than once. For almost all computations, R uses "double-precision floating-point" arithmetic, as most other systems do. What this means is that R can represent numbers up to about $\pm 1.79 \times 10^{\pm 308}$ with at least some accuracy. However, double precision is not exact. Consider this example, in which we multiply together the numbers (1/49) and 49.

```
> 1/49 * 49
[1] 1                          # as expected
> 1 - (1/49 * 49)
[1] 1.110223e-16
> (49 * 1/49) == (1/49 * 49) # should be TRUE
[1] FALSE
```

The first computation shows the "expected" product of (1/49) and 49 – the value 1. In fact, though, the second computation shows that this product is not exactly 1; it differs from 1 by a tiny amount that we might call "floating-point error." That amount was so small that it wasn't displayed in the first computation, according to R's default display conditions. (The command `print(1/49 * 49, digits = 16)` will reveal that this product is computed as a number very slightly less than 1.) This is not a bug in R; it's a statement about the way double-precision floating-point arithmetic works, analogous to the way that in ordinary arithmetic, the number 0.333333… is not quite 1/3. The final computation shows the practical effect of this: if you compare two floating-point values directly, they might be recorded as being different just because of floating-point error. You will need to be aware of this when you compare the results of doing the same computation in two different ways.

Significant Digits

In the above-mentioned example, we saw how R printed `1` even though the number in question was slightly different. While R's computations use

double-precision floating point, its display will generally print a smaller number of digits than are available. Moreover, R formats outputs in a neat way, so that typing `2.00` produces `2`, but typing `2.01` prints out as `2.01`. These formatting choices are most noticeable when many values are being shown. The display that R chooses does not affect the precision with which it does calculations. Of course you can force R to round off the results of its calculation; we discuss formatting, rounding, and scientific notation in Chapter 4.

Character Strings

We will spend a lot of time in this book handling text or character data, data in the form of letters such as `"Oakland"` or `"Missing"`. Sometimes, as is common, we will call a set of characters a *string*. In R, strings are enclosed by quotation marks, and either the double-quotation mark `"` or the single one `'` can be used. A string delineated by single-quotation marks is converted into the other kind. The two kinds of quotation marks make it possible to insert a quote into a string, such as this: `"She said 'No.' "` (If you typed `"She said "No." "`, you would see R produce an error.) If you type `'She said "No." '`, the outside quotes are converted to double quotes. Then, since there are double quotes on the inside, too, those interior quotation marks are "protected" by preceding them with the backslash character. The result is converted into `"She said \"No.\" "`

This idea of "protecting" certain special characters goes beyond quotation marks. The character that marks the end of a line of text is called "new-line" and is written as `\n`, backslash followed by `n`. Typing this character requires two keystrokes, but it counts as only one character. In general, special characters are "protected" by the backslash characters. Besides the quotation mark and the new-line, the important special characters are `\t`, the tab, and `\\`, the backslash itself. The backslash also serves to introduce strings in special formats, such as hexadecimal (e.g., `"\xb1"` produces the character with hexadecimal value `b1`, which displays as the plus-minus sign, ±) or Unicode (e.g., `"\U20ac"` uses Unicode to display the Euro currency symbol). We talk much more about text in general and Unicode in particular in Chapter 4.

1.4 Running an R Session

Once you start using R you may find yourself using it for lots of different projects. Although this is partly a matter of taste, we find it useful to keep separate sets of data for separate projects. In this section, we describe where R keeps your data, and some other aspects of R with which you will need to be familiar.

1.4.1 Where Your Data Is Stored

When you start R, you start it in a *working directory*, and this directory forms the starting point for where R looks for, and stores, data. For example, typing `list.files()` will list all of the files in your working directory. When you quit R and save the workspace, a file with all of your R objects will be created in that same directory. This file is named `.RData`. The leading dot in the name is important, because some terminal programs, such as the "bash" command interpreter, do not by default list files whose names start with a dot. We don't recommend changing the name of the `.RData` file.

This provides a natural mechanism for project management. To prepare for a new project on a system with a command-line interface, just create a new directory and start R from there (see "starting R" above). On systems with desktop icons, copy an existing R icon, edit the properties to point to the new directory, and add the project name to the icon. The details of this operation will depend on your operating system. In this way, you can keep the different `.RData` files for your different projects separate.

When you start R, it will use an existing `.RData` file if there is one in the working directory, or create a new, empty one if there is not. Often we have a certain number of objects from earlier projects that we want in the new project. There are two mechanisms for acquiring those existing R objects. In one case, we literally copy all the objects from another `.RData` in a different project's directory into the existing workspace, using the `load()` function. This can be dangerous because objects being copied will over-write existing ones with the same names. A second mechanism uses `attach()`, which puts the other `.RData` on the "search path." The search path is a list of places where R looks for objects when you mention them. You can examine your current search path with the `search()` command. The first entry on the search path is the current `.RData` file (although it carries the confusing name `.GlobalEnv`); most of the other entries on the search path are put there by R itself. When you use a name such as `pi`, R looks for that object in your workspace, and then in each of the packages or directories named in the search path until it finds one by that name. You can attach other `.RData` files anywhere in the search path, except in the first position; usually we put them into position two so that they are searched right after the local workspace. We talk more about getting data into and out of R in Chapter 6.

1.4.2 Options

R maintains a list of what it calls "options," which describe aspects of your interaction with it. For example, one option sets the text editor that R calls when you edit a function, one describes how much memory is set aside for R, one lets you change the prompt character from its default, and so on. Generally, we

find the default values reasonable, but the help for the `options()` function describes the possible values and running `options()` shows you the current ones. Changes to the options last only for this R session. Section 3.3.2 shows an example of setting one of the options.

1.4.3 Scripts

Most of the work we do with R is interactive – that is, we issue commands and wait for R's response. This use of R is best when we are exploring data and developing approaches to handling and modeling it. As we develop sets of commands for a particular project, we can combine these into "scripts," which are simply files full of commands. Having a set of commands together allows us to execute them in exactly the same way every time, and it allows us to add comments and other notes that will be useful to us and to other users whom we share the code with. This approach, while still interactive, is best when we have developed an approach and want to use it repeatedly. Scripts also provide a natural mechanism for project management: often we start with an empty workspace and use scripts to populate the workspace by reading and preparing data, loading from other sources, or attaching other directories, before starting on the modeling steps.

R can also be run in batch mode – that is, it can start, run a single set of commands, and then stop. This approach can be used when the same task needs to be performed repeatedly, perhaps on different data – say, every day to process data gathered overnight. We talk about scripts and batch use of R in Chapter 5.

1.4.4 R Packages

A *package* is a set of functions (and maybe data and other stuff too) that provides an extension to R. R comes with a set of packages, some of which are automatically placed onto the search path, and others of which are not. If a package is present on your computer but not in your search path, you can access (or "load") it with the `library()` or `require()` command (these two differ only in how they react if a package cannot be found). A package only needs to be loaded once per R session, but when you re-start R you will need to re-load packages. There are also thousands of additional packages that have been contributed by R users that can be found on the Internet, primarily at the main repository at `cran.r-project.org` and its mirror sites. If your computer is connected to the Internet, you can install a package (if you know its name) with the `install.packages()` command. If that works, the package will still need to be loaded with the `library()` command. If your computer is not connected to the Internet, you can still install packages from a disk file if one is available. Most of the code in this book requires no additional packages, although in some cases we will point out cases where additional packages make particular tasks easier, more efficient, or, in rare cases, possible.

It is possible to force certain packages to be loaded whenever you start R. When we anticipate needing a package, our preference is to include a call to `library()` or `require()` inside our scripts.

1.4.5 RStudio and Other GUIs

The "look" of R depends on your operating system. At its most basic – and we often see this when we are connecting to remote servers – R consists only of a command line. On the most popular platforms – Windows and OS X – running R produces a graphical user interface, or GUI. This is a set of windows containing a number of menu items giving selections, or buttons that help you perform common tasks. Most of the GUI, though, consists of the console. A few enhanced GUIs are available. Perhaps the most widely used among these is RStudio (RStudio Team, 2015), a development environment that includes a console window, a set of script window tabs, and better handling of multiple graphics windows. RStudio comes in free and paid versions for all operating systems and is available from its maker at `rstudio.com`. We have found that many of our students prefer the more interactive, perhaps more modern feel of RStudio to the standard R interface – but underneath, the R language is exactly the same.

1.4.6 Locales and Character Sets

R is essentially the same program whether you run it on Windows, OS X, or Linux. (There are minor differences in the way you access external files and in some low-level technical functions that will not be relevant in data cleaning.) In particular, R is an English-language program, so a "for" loop is always indicated by `for()`. Speakers of many languages can arrange to have error messages delivered in their language, if this ability is configured at the time R is installed – see the help for the `Sys.setenv()` function and for "environment variables."

Even though R is in English, it is possible to set the "locale" of R. This allows you to change the way that R does things such as format currency values. English speakers use the dot as the decimal separator and the comma to set off thousands from hundreds, but many Europeans use those two characters in reverse. Other locale settings affect the abbreviations in use for days of the week and months of the year. We discuss some of these in Chapter 3, but one important one to note here is the "collation" setting. This describes how R sorts alphabetical items. Under the usual choices on Windows and OS X, lower- and upper-case letters are sorted together, so that "a" precedes "A" in alphabetical order, but both precede "b." To continue an earlier example, this ensures that `"apple" < "banana"` and `"apple" < "Banana"` are both `TRUE`. However, on some Linux systems the so-called "C" collation sequence is used. In that scheme, all the upper-case letters come before

any of the lower-case ones – so that `"apple" < "banana"` is `TRUE`, but `"apple" < "Banana"` is `FALSE`. Moreover, as the help for `Comparison` points out, "in Estonian, `Z` comes between `S` and `T`." You have to be aware of your both locale and the relevant language whenever you compare strings.

Another aspect of character handling is the use of different character sets. Text in non-Roman languages such as Hebrew or Korean requires some special considerations. We discuss these at some length in Chapter 4.

1.5 Getting Help

R has a number of ways of getting help to you. "Help" can mean information about the specific syntax of individual R commands, about putting the pieces of R together in programs, or about the details of the various statistical models and tools that R provides. In this section, we describe some of the resources available to help you learn about R.

1.5.1 At the Command Line

The most basic help is provided at the command line, through the commands `help()`, `?` and `help.search()`. The first two commands act identically and will be most useful when you need information on a particular R function or operator whose name you know. In most cases, the argument doesn't need to be in quotation marks, though it may be – so `help(matrix)` or `?"matrix"` both bring up a page about some matrix functions. Quotation marks will be required when looking for help on some elements of the R language – so `?"for"` gives the help page for the `for` looping term and `help("==")` produces the page on comparison operators. The `help.search()` command is useful when the subject, rather than the name, is known; this command opens a window (depending on your operating system) that gives links to associated R objects. A related command is the `apropos()` function, which takes a character argument (as in `apropos("matrix")`) and returns a vector of names of objects containing that string (in this example, every object with `matrix` in its name). A final piece of command-line help is provided by the `args()` function, which takes a function and displays the set of arguments expected by, and default values provided by that function.

1.5.2 The Online Manuals

When you install R, you are given the opportunity to install the online manuals with it. These manuals are generally correct and complete, but they are intended as references, and are not always useful as tutorials.

1.5.3 On the Internet

The main page for the R project is `r-project.org`. This is the central repository for R and its documentation. If you are interested in participating in a community of R users, you might consider joining one of the mailing lists, which you can find under `mail.html` at that page.

R is very popular and there are lots and lots of blogs, pages, and other web documents that address R and solve specific problems. Your favorite Internet search engine will be able to find dozens of these.

1.5.4 Further Reading

A lot of documentation comes with R when you install in the usual way. You can find a list of these manuals under Help | Manuals in Windows, or Help | R Help on OS X, or with `help.start()`. The "Introduction to R" manual is a good place to start.

The book "The Art of R Programming" (Matloff, 2011) is a nice tour of many R features ranging from beginning to advanced. As its name suggests, the emphasis is on writing powerful and efficient R programs. Many other books introduce the use of R, or describe its application in specific fields such as economics or genomics. The `r-project` website has a list of over 150 books using R. As we mentioned earlier, that site also maintains mailing lists for interested users, and a quick web search will reveal scores of blogs and web pages devoted to R and to answering R questions.

The recent book by Wickham and Grolemond (2016) describes those authors' approach to not only data cleaning but a set of additional tasks, including visualization and modeling, which we think of as beyond the scope of data acquisition and cleaning. That approach requires an entire set of tools from packages outside R – although they come conveniently bundled together – as well as a new vocabulary. This ecosystem has its adherents, but we prefer to use base R where possible.

1.6 How to Use This Book

1.6.1 Syntax and Conventions in This Book

We reproduce a lot of R code in this book. R code is indicated in a fixed-width font `like this`. Since R is case-sensitive, our text will exactly match what is typed into R – except that in the prose we capitalize letters of R objects if they appear at the beginning of sentences. Inside a paragraph, or when we want to show a sequence of commands, we reproduce exactly what we type, like this:

`sqrt(pi)`. When we also want to show what R returns, the code will be shown with the prompt and the literal R output, like this:

```
> sqrt (pi)
[1] 1.772454
```

Unlike the example in the "top ten quick fact" #1, we suppress the continuation prompt +, so that it is not confused with the ordinary plus sign.

There are several different schemes for formatting code that you can find described on the Internet, and they do not always agree. To us the most important rule is to make your code easy to read. This means, first, use spacing and indenting in a helpful and consistent way, and second, add plenty of comments to help the reader. There is always a temptation to write code as quickly as possible, with an eye toward worrying about neatness later. Resist that temptation! Code is for sharing and for re-use.

On a lighter note, we know that the word "data" originated as the plural of the singular "datum," but it has long been permitted to construe "data" in the singular, and we do that in this book. You will find us saying "the data is..." rather than "are." This is intentional.

1.6.2 The Chapters

In order to use R wisely, you have to understand what data looks like to R. The following three chapters describe the sorts of data that R recognizes, and how to manipulate R's objects. We start by describing vectors, the simplest form of data in R, in Chapter 2. This chapter describes the common types of vectors, the different ways to extract subsets from them, and how to change values in vectors. It also describes how R stores missing values, an integral part of almost every data cleaning problem. The chapter concludes with a look at the important `table()` function and some of the basic operations on vectors – sorting, identifying duplicates, computing unions and intersections of sets, and so on.

Chapter 3 describes more complicated data structures: matrices, lists, and finally data frames. Understanding how data frames work is critical to using R intelligently. We defer until this chapter discussion of how R handles times and dates, because part of that discussion requires an understanding of lists.

The final data chapter, Chapter 4, discusses the last important data type – text or character data. Text data is stored in vectors and data frames such as other kinds, but there are a number of operations specific to text. This chapter describes how to manipulate text in R – changing case, extracting and assembling pieces of strings, formatting numbers into strings, and so on. One important topic is *regular expressions*, a set of tools for finding strings that contain a pattern of characters. This chapter also discusses the UTF-8 system of encoding non-Roman alphabets such as Greek or Chinese and R's concept of *factors*, which are important in modeling but often cause problems during the data cleaning process.

Chapter 5 discusses two types of tool used to automate computations in R: functions and scripts. These different, but related, tools, will be part of every analysis you ever do, so you should understand how to construct them intelligently. We also look briefly at "shell scripts," which are a special sort of script that let you run R in batch, rather than interactive mode, and discuss some of the tools available in R for debugging.

This is a book about cleaning data, but the data to be cleaned needs to come from somewhere. Chapter 6 describes the different ways to bring data into R: from other R sessions, from spreadsheet-like text files, from relational databases, and so on. We describe two of the formats in which data is commonly found in modern applications: XML and JSON. We also describe how to acquire data programmatically from web pages.

Chapter 7 takes a bigger view of the data cleaning process. While the earlier chapters focus on the nuts and bolts of R as they relate to data cleaning, this chapter describes the sort of challenges in a real-life data cleaning project. We talk about how to combine data from different sources and give examples of the sort of anomalies that you have to expect in dealing with real data. In almost every case you will have to rely on judgment, rather than just on a cookbook of techniques. We spend some time discussing the role of judgment on data cleaning.

The Exercise

The culmination of the book is the data cleaning exercise presented in Chapter 8. This chapter presents a complicated data acquisition and cleaning problem that, while artificial, reflects many of the problems and challenges we have seen over our years of real-life data handling experience. If you can find your way through to the end of the exercise, we expect that you will be well prepared to handle the data the real world sends your way.

Critical Data Handling Tools

In every chapter, we have set aside the final section to recap commands and tools we think are particularly important when it comes to data handling and manipulation. If you can master the use of these tools, and apply them wisely, you can reduce the risk of missing important information in your data.

The Code

All of the code reproduced in this book appears in scripts in the `cleaningBook` package you can download from the CRAN website. You can open these scripts in R and run the code from there – although since most examples are very short, we suggest that you consider typing them in yourself, to get a feel for the R language.