Advanced R

Data Programming and the Cloud

—

Matt Wiley
Joshua F. Wiley

Apress®

Advanced R

Data Programming and the Cloud

Matt Wiley
Joshua F. Wiley

Apress®

Advanced R: Data Programming and the Cloud

Matt Wiley
Elkhart Group Ltd. & Victoria College
Columbia City, Indiana
USA

Joshua F. Wiley
Elkhart Group Ltd. & Victoria College
Columbia City, Indiana
USA

ISBN-13 (pbk): 978-1-4842-2076-4
ISBN-13 (electronic): 978-1-4842-2077-1
DOI 10.1007/978-1-4842-2077-1

Library of Congress Control Number: 2016959581

Managing Director: Welmoed Spahr
Lead Editor: Steve Anglin
Technical Reviewer: Andrew Moskowitz
Editorial Board: Steve Anglin, Pramila Balan, Laura Berendson, Aaron Black, Louise Corrigan, Jonathan Gennick, Robert Hutchinson, Celestin Suresh John, Nikhil Karkal, James Markham, Susan McDermott, Matthew Moodie, Natalie Pao, Gwenan Spearing
Coordinating Editor: Mark Powers
Copy Editor: Sharon Wilkey
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a Delaware corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text are available to readers at www.apress.com. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/ . Readers can also access source code at SpringerLink in the Supplementary Material section for each chapter.

Printed on acid-free paper

To Family.

Contents at a Glance

Contents

About the Authors

Matt Wiley is a tenured, associate professor of mathematics with awards in both mathematics education and honor student engagement. He earned degrees in pure mathematics, computer science, and business administration through the University of California and Texas A&M systems. He serves as director for Victoria College's quality enhancement plan and managing partner at Elkhart Group Limited, a statistical consultancy. With programming experience in R, C++, Ruby, Fortran, and JavaScript, he has always found ways to meld his passion for writing with his joy of logical problem solving and data science. From the boardroom to the classroom, Matt enjoys finding dynamic ways to partner with interdisciplinary and diverse teams to make complex ideas and projects understandable and solvable.

Joshua F. Wiley is a lecturer in the Monash Institute for Cognitive and Clinical Neurosciences and School of Psychological Sciences at Monash University and a senior partner at Elkhart Group Limited, a statistical consultancy. He earned his PhD from the University of California, Los Angeles, and his research focuses on using advanced quantitative methods to understand the complex interplays of psychological, social, and physiological processes in relation to psychological and physical health. In statistics and data science, Joshua focuses on biostatistics and is interested in reproducible research and graphical displays of data and statistical models. Through consulting at Elkhart Group Limited and former work at the UCLA Statistical Consulting Group, he has supported a wide array of clients ranging from graduate students, to experienced researchers, to biotechnology companies. He also develops or co-develops a number of R packages including `varian`, a package to conduct Bayesian scale-location structural equation models, and `MplusAutomation`, a popular package that links R to the commercial Mplus software.

About the Technical Reviewer

Andrew Moskowitz is a doctoral candidate in quantitative psychology at the University of California, Los Angeles, and a self-employed statistical consultant. His quantitative research focuses mainly on hypothesis testing and effect sizes in mixed-effects models. While at UCLA, Andrew has collaborated with a number of faculty, students, and enterprises to help them derive meaning from data across an array of fields ranging from psychological services and health care delivery to marketing.

Acknowledgments

We would like to profusely thank our technical reviewer, Andrew Moskowitz. Through direct comments in chapters, e-mails about proper explanations, and Skype calls, Andrew gave us a lot of thoughtful feedback. If our readers feel that any portion explains a technique well, that is thanks to his efforts; the errors of course remain ours alone.

Mark Powers has been extraordinarily kind to us, and this book would not be here without his advocacy and support. Steve Anglin also deserves thanks for working with us to start this project. Truly, if you look at the very front of this book, there is an entire team at Apress who deserve rich and warm thanks.

Introduction

R has become one of the most popular programming languages in an era where data science is increasingly prevalent. As R and data science have become more mainstream, there is a growing number of R users without dedicated training in statistical computing or data science, and thus a growing demand for books and resources to bridge the gap between applied users who may have only an introductory background in statistics or programming and advanced and sophisticated data analytics. This book focuses on how to use advanced programming in R to speed up everyday tasks in data analysis and data science. This book is also unique in its coverage of how to set up R in the cloud and generate dynamic reports for analyses that are regularly repeated, such as monthly analysis of company sales or quarterly analysis of student grades, enrollment, and dropout numbers in schools with projections for future enrollment rates.

Chapters 1 through 6 focus on more advanced programming techniques than the Apress offering of *Beginning R*.

Chapters 7–10 develop powerful data management measures including the exciting and (comparatively) new `data.table`.

From here, we delve into the modern (and slightly edgy) world of cloud computing with R. From the ground up, we walk you through getting R started on an Amazon cloud in chapters 11–14.

Finally, Chapter 15 provides you with solid techniques in dynamic documents and reports.

CHAPTER 1

Programming Basics

As with most languages, more advanced usage requires delving into the underlying structure. This chapter covers such programming basics, and this first section of the book (through Chapter 6), develops some advanced programming techniques. We start with R's basic building blocks, which create our foundation for programming, data management, and cloud analytics.

Before we dig too deeply into R, some general principles to follow may well be in order. First, experimentation is good. It is much more powerful to learn hands-on than it is simply to read. Download the source files that come with this text, and try new things!

Second, it can help quite a bit to become familiar with the ? function. Simply type ? immediately followed by text in your R console to call up help of some kind. We cover more on functions later, but this is too useful to ignore until that time.

Finally, just before we dive into the real reason you bought this book, a word of caution: this is an applied text. There may be topics and areas of R we skip or ignore. While we, the authors, like to imagine this is due to careful pruning of ideas, it may well be due to ignorance. There are likely other ways to perform these tasks or additional good topics to learn. Our goal is to get you up and running as quickly as possible toward some useful skills. Good luck!

Advanced R Software Choices

This book is written for advanced users of the R language. We should note that for most of our examples, we continue using RStudio (`www.rstudio.com/products/rstudio/download/`) as in *Beginning R: An Introduction to Statistical Programming* (Apress, 2015). We also assume you are using a Microsoft Windows (`www.microsoft.com`) operating system, except for the later chapters, where we delve into using R in the cloud via Ubuntu (`www.ubuntu.com`). What is different is the underlying R distribution.

We are going to use Microsoft R Open (MRO), which is fully aligned with the current version(s) of R. This provides performance enhancements that happen behind the scenes. We also use Intel Math Kernel Library (Intel MKL), which is available for download at the same site as MRO (`https://mran.microsoft.com/download/`). In fact, as this book goes to print, these two software programs combined in their latest release. It would be wonderful if that trend continues. These downloads are very straightforward, and we anticipate that our readers, familiar with using R and RStudio already, find this a seamless installation. On Windows (and Linux-based operating systems), the MKL replaces the default linear algebra system with an optimized system and allows implicit parallel processing for linear algebra operations, such as matrix multiplication and decomposition that are used in many statistical algorithms.

Electronic supplementary material The online version of this chapter (doi: 10.1007/978-1-4842-2077-1) contains supplementary material, which is available to authorized users.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_1

In case it is not already, you also need Java installed. We used Java Version 8 Update 91 for 64 bit in this book. Java may be downloaded at www.oracle.com/technetwork/java/javase/; specifically, get the Java Development Kit (JDK).

While these choices may have minor consequences, our goal is to provide universal guidance that remains true enough regardless of environmental specifics. Nevertheless, some packages and prebuilt functions on occasion have quirks. We turn our attention to ensuring that you can readily reproduce our results.

Reproducing Results

One useful feature of R is the abundance of packages written by experts worldwide. This is also potentially the Achilles' heel of using R: from the version of R itself to the version of particular packages, lots of code specifics are in flux. Your code has the potential to not work from day to day, let alone our code written months before this book was published. To solve this, we use the Revolution Analytics `checkpoint` package (Microsoft Corporation, 2016), which uses server-stored snapshots from the Comprehensive R Archive Network (CRAN) to "lock" our code to a specific version and date. To learn the technical specifics of how this is done, visit the link in the "References" section at the end of this chapter. We'll get you started with the basics.

For this book, we used R version 3.3.1, Bug in Your Hair, along with Windows 10 Professional x64. As this version moves from the current version to historical, CRAN maintains an archive of past releases. Thus, the `checkpoint` package has ready access to previous versions of R, and indeed all packages. What you need to do is add the following code to the top of your Chapter 1 R file in your project directory:

```
## uncomment to install the checkpoint package
## install.packages("checkpoint")
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(data.table)
```

We place all library calls at the start of each chapter's project file, after the call to the `checkpoint` library. By including the date of September 4, 2016, we ensure that the latest version of all packages up to that cutoff is installed and run by `checkpoint`. The first time it is run, after asking permission, `checkpoint` creates a folder to host the needed versions of the packages used. Thus, as long as you start each chapter's code file with the correct library calls, you use the same versions of the packages we use.

Types of Objects

First of all, we need things to build our language, and in R, these are called *objects*. We start with five very common types of objects.

Logical objects take on just two values: `TRUE` or `FALSE`. Computers are binary machines, and data often may be recorded and modeled in an all-or-nothing world. These logical values can be helpful, where `TRUE` has a value of `1`, and `FALSE` has a value of `0`:

```
TRUE
[1] TRUE
FALSE
[1] FALSE
```

As you may remember from the quickly muttered comments of your algebra professor, there are many types, or flavors, of numbers. Whole numbers, which include zero as well as negative values, are called *integers*. In set notation, {...,-2, -1, 0, 1, 2, ...}, these numbers are helpful for headcounts or other indexes (as well as other things, naturally). In R, integers have the capital `L` suffix. If decimal numbers are needed, then *double* numeric objects are in order. These are the numbers suited for even-ratio data types. *Complex numbers* have useful properties as well and are understood precisely as you might expect, with an `i` suffix on the imaginary portion. R is quite friendly in using all of these numbers, and you simply type in the desired numbers (remember to add the `L` or `i` suffix as needed):

```
42L
[1] 42
1.5
[1] 1.5
2+3i
[1] 2+3i
```

Nominal-level data may be stored via the `character` class and is designated with quotation marks:

```
"a" ## character
[1] "a"
```

Of course, numerical data may have missing values. These missing values are of the type that the rest of the data in that set would be (we discuss data storage shortly). Nevertheless, it can be helpful to know how to hand-code logical, integer, double, complex, or character missing values:

```
NA
[1] NA
NA_integer_
[1] NA
NA_real_
[1] NA
NA_character_
[1] NA
NA_complex_
[1] NA
```

Factors are a special kind of object, not so useful for general programming, but used a fair amount in statistics. A factor variable indicates that a variable should be treated discretely. Factors are stored as integers, with labels to indicate the original value:

```
factor(1:3)
[1] 1 2 3
Levels: 1 2 3
factor(c("a", "b", "c"))
[1] a b c
Levels: a b c
factor(letters[1:3])
[1] a b c
Levels: a b c
```

We turn now to data structures, which can store objects of the types we have discussed (and of course more). A *vector* is a relatively simple data storage object. A simple way to create a vector is with the concatenate function `c()`:

```
c(1, 2, 3)
[1] 1 2 3
```

Just as in mathematics, a *scalar* is a vector of just length 1. Toward the opposite end of the continuum, a *matrix* is a vector with dimensions for both rows and columns. Notice the way the matrix is populated with the numbers 1 through 6, counting down each column:

```
c(1)
[1] 1

matrix(c(1:6), nrow = 3, ncol = 2)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

All vectors, be they scalar, vector, or matrix, can have only one data type (for example, integer, logical, or complex). If more than one type of data is needed, it may make sense to store the data in a list. A `list` is a vector of objects, in which each element of the list may be a different type. In the following example, we build a list that has character, vector, and matrix elements:

```
list(
+   c("a"),
+   c(1, 2, 3),
+   matrix(c(1:6), nrow = 3, ncol = 2)
+   )
[[1]]
[1] "a"

[[2]]
[1] 1 2 3

[[3]]
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
```

A particular type of list is the *data frame*, in which each element of the list is identical in length (although not necessarily in object type). Take a look at the following instructive examples with output:

```
data.frame(1:3, 4:6)
  X1.3 X4.6
1    1    4
2    2    5
3    3    6
```

```
## using non equal length objects causes problems
data.frame( 1:3, 4:5)
Error in data.frame(1:3, 4:5) :
  arguments imply differing number of rows: 3, 2

data.frame( 1:3, letters[1:3])
  X1.3 letters.1.3.
1    1            a
2    2            b
3    3            c
```

Because of their superior speed, we use data table objects in R from the `data.table` package. Data tables are similar to data frames, but are designed to be more memory efficient and faster. Even though we recommend data tables, we show some examples with data frames as well because when you work with R, many other people's code includes data frames, and indeed data tables inherit many methods from data frames.

```
library(data.table)
data.table( 1:3, 4:6)
   V1 V2
1:  1  4
2:  2  5
3:  3  6
```

Having explored several types of objects, we turn our attention to ways of manipulating those objects with operators and functions.

Base Operators and Functions

Objects are not enough for a language; some things require actions. *Operators* and *functions* are the verbs of the programming world. We start with assignment, which can be done in two ways. Much like written languages, more-elegant turns of phrase can be more helpful than simpler prose. So although = and <- are both assignment operators and do the same thing, because = is used within functions to set arguments, we recommend for clarity's sake to use <- for general assignment. We nevertheless demonstrate both assignment techniques. Assignments allow objects to be given sensible names; this can significantly enhance code readability (for your future self as well as for other users).

In addition to assigning names to variables, you can check specifics by using functions. Functions in R take the general format of function name, followed by parentheses, with input inside the parentheses, and then R provides output. Here are examples:

```
x <- 5
y = 3
x
[1] 5
y
[1] 3

is.integer(x)
[1] FALSE
```

```
is.double(y)
[1] TRUE

is.vector(x)
[1] TRUE
```

Once an object is assigned, you can access specific object elements by using brackets. Most computer languages start their indexing at either 0 or 1. R starts indexing at 1. Also, note that you can readily change old assignments with little trouble and no warning; it is wise to watch names cautiously and comment code carefully.

```
x <- c("a", "b", "c")
x[1]
[1] "a"

is.vector(x)
[1] TRUE

is.vector(x[1])
[1] TRUE

is.character(x[1])
[1] TRUE
```

While a vector may take only a single index, more-complex structures require more indices. For the matrix you met earlier, the first index is the row, and the second is for column position. Notice that after building a matrix and assigning it, there are many ways to access various combinations of elements. This process of accessing just some of the elements is sometimes called *subsetting*:

```
x2 <- matrix(c(1:6), nrow = 3, ncol = 2)
x2
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

x2[1, 2] ## row 1, column 2
[1] 4

x2[1, ] ## all row 1
[1] 1 4

x2[, 1] ## all column 1
[1] 1 2 3

x2[c(1, 2), ] ## rows 1 and 2
     [,1] [,2]
[1,]    1    4
[2,]    2    5
```

```
x2[c(1, 3), ] ## rows 1 and 3
     [,1] [,2]
[1,]    1    4
[2,]    3    6

x[-2] ## drop element two
[1] "a" "c"

x2[, -2] ## drop column two
[1] 1 2 3

x2[-1, ] ## drop row 1
     [,1] [,2]
[1,]    2    5
[2,]    3    6

is.vector(x2)
[1] FALSE

is.matrix(x2)
[1] TRUE
```

Accessing and subsetting lists is perhaps a trifle more complex, yet all the more essential to learn and master for later techniques. A single index in a single bracket returns the entire element at that spot (recall that for a list, each element may be a vector or just a single object). Using double brackets returns the object within that element of the list—nothing more.

Thus, the following code is, in fact, a vector with the element a inside. Again, using the data-type-checking functions can be helpful in learning how to interpret various pieces of code.

```
y <- list( c("a"), c(1:3))
y[1]
[[1]]
[1] "a"

is.vector(y[1])
[1] TRUE

is.list(y[1])
[1] TRUE

is.character(y[1])
[1] FALSE
```

Contrast that with this code, which is simply the element `a`:

```
y[[1]]
[1] "a"

is.vector(y[[1]])
[1] TRUE
```

```
is.list(y[[1]])
[1] FALSE

is.character(y[[1]])
[1] TRUE
```

You can, in fact, chain brackets together, so the second element of the list (a vector with the numbers 1 through 3) can be accessed, and then, within that vector, the third element can be accessed:

```
 y[[2]][3]
 [1] 3
```

Brackets almost always work, depending on the type of object, but there may be additional ways to access components. Named data frames and lists can use the $ operator. Notice in the following code how the bracket or dollar sign ends up being equivalent:

```
x3 <- data.frame( A = 1:3, B = 4:6)
y2 <- list( C = c("a"), D = c(1, 2, 3))

x3$A
[1] 1 2 3
y2$C
[1] "a"
x3[["A"]]
[1] 1 2 3
y2[["C"]]
[1] "a"
```

Notice that although both data frames and lists are both lists, neither is a matrix:

```
is.list(x3)
[1] TRUE

is.list(y2)
[1] TRUE

is.matrix(x3)
[1] FALSE

is.matrix(y2)
[1] FALSE
```

Moreover, despite not being matrices, because of their special nature (that is, all elements have equal length), data frames and data tables can be indexed similarly to matrices:

```
x3[1, 1]
[1] 1

x3[1, ]
  A B
1 1 4
```

```
x3[, 1]
[1] 1 2 3
```

Any named object can be indexed by using the names rather than the positional numbers, provided those names have been set:

```
x3[1, "A"]
[1] 1

x3[, "A"]
[1] 1 2 3
```

This applies to both column and row names, and these names can be established after building the matrix:

```
rownames(x3) <- c("first", "second", "third")


x3["second", "B"]
[1] 5
```

Data tables use a slightly different approach. Selecting rows works almost identically but selecting columns does not require quotes. Additionally, you can select multiples by name without quotes by using the `.()` operator. Should you need to use quotes, the data table can be accessed by using the option `with = FALSE` such as follows:

```
x4 <- data.table( A = 1:3, B = 4:6)


x4[1, ]
   A B
1: 1 4

x4[, A]
[1] 1 2 3
x4[1, A]
[1] 1

x4[1:2, .(A, B)]
   A B
1: 1 4
2: 2 5

x4[1, "A", with = FALSE]
   A
1: 1
```

Technically, the bracket operators are functions. Although they're not used as functions, they can be. Most functions are named, but the brackets are a particular case and require using single quotes in the regular function format, as in the following example:

```
`[`(x, 1)
[1] "a"

`[`(x3, "second", "A")
[1] 2
```

Although we have been using the `is.datatype()` function to better illustrate what an object is, you can do more. Specifically, you can check whether a value is missing an element by using the `is.na()` function:

```
is.na(NA) ## works
[1] TRUE
```

Of course, the preceding code snippet usually has a vector or matrix element argument whose populated status is up for debate. Our last (for now) exploratory function is the `inherits()` function. It is helpful when no `is.class()` function exists, which can occur when specific classes outside the core ones you have seen presented so far are developed:

```
inherits(x3, "data.frame")
[1] TRUE
inherits(x2, "matrix")
[1] TRUE
```

You can also force lower types into higher types. This *coercion* can be helpful but may have unintended consequences. It can be particularly risky if you have a more advanced data object being coerced to a lesser type (pay close attention to the attempt to coerce an integer).

```
as.integer(3.8)
[1] 3

as.character(3)
[1] "3"

as.numeric(3)
[1] 3

as.complex(3)
[1] 3+0i

as.factor(3)
[1] 3
Levels: 3

as.matrix(3)
     [,1]
[1,]    3

as.data.frame(3)
  3
1 3
```

```
as.list(3)
[[1]]
[1] 3

> as.logical("a")
[1] NA

as.logical(3)
[1] TRUE

as.numeric("a")
[1] NA
Warning message:
NAs introduced by coercion
```

Coercion can be helpful. All the same, it must be used cautiously. Before you move on from this section, if any of this is new, be sure to experiment with different inputs than the ones we tried in the preceding example! Experimenting never hurts, and it can be a powerful way to learn.

Let's turn our attention now to mathematical and logical operators and functions.

Mathematical Operators and Functions

Several operators can be used for comparison. These will be helpful later, once we get into loops and building our own functions. Equally useful are symbolic logic forms. We start with some basic comparisons and admit to a strange predilection for the number 4:

```
4 > 4
[1] FALSE
4 >= 4
[1] TRUE
4 < 4
[1] FALSE
4 <= 4
[1] TRUE
4 == 4
[1] TRUE
4 != 4
[1] FALSE
```

It is sensible now to mention that although the preceding code may be helpful, often numbers differ from one another only slightly—particularly in the programming environment, which relies on the computer representation of floating-point (irrational) numbers. Therefore, we often check that things are close within a tolerance:

```
all.equal(1, 1.00000002, tolerance = .00001)
[1] TRUE
```

In symbolic logic, AND as well as OR are useful comparisons between two objects. In R, we use & for AND, as well as | for OR. Complex logic tests can be constructed from these simple structures:

```
TRUE | FALSE
[1] TRUE
FALSE | TRUE
[1] TRUE
TRUE & TRUE
[1] TRUE
TRUE & FALSE
[1] FALSE
```

All of the logic tests mentioned so far apply just as well to vectors as they apply to single objects:

```
1:3 >= 3:1
[1] FALSE  TRUE  TRUE

c(TRUE, TRUE) | c(TRUE, FALSE)
[1] TRUE TRUE

c(TRUE, TRUE) & c(TRUE, FALSE)
[1]  TRUE FALSE
```

If you want only a single response, such as for if-else flow control, you can use && or ||, which stop evaluating as soon as they have determined the final result. Work through the following code and output carefully:

```
W
Error: object 'W' not found

TRUE | W
Error: object 'W' not found

TRUE || W
[1] TRUE

W || TRUE
Error: object 'W' not found

FALSE & W

Error: object 'W' not found

FALSE && W
[1] FALSE
```

Note that the double operators are not, in fact, vectorized. They simply use the first element of any vectors:

```
c(TRUE, TRUE) || c(TRUE, FALSE)
[1] TRUE
```

```
c(TRUE, TRUE) && c(TRUE, FALSE)
[1] TRUE
```

The `any()` and `all()` functions are helpful as well in these contexts for similar reasons:

```
any(c(TRUE, FALSE, FALSE))
[1] TRUE

all(c(TRUE, FALSE, TRUE))
[1] FALSE

all(c(TRUE, TRUE, TRUE))
[1] TRUE
```

We turn our attention now to mathematical, rather than logical, operators. R is powerful mathematically and can perform most mathematical calculations. So although we introduce some functions, we are leaving many out of the mix. For more details, `?Arithmetic` can be your friend. It is (as always) important to be aware of the way computers perform mathematical calculations. Being able to code bespoke solutions directly is powerful, yet with the freedom to customize comes a corresponding amount of responsibility. Take a careful look at the following mathematical operations (which can behave differently than expected because of implementation choices):

```
3 + 3
[1] 6
3 - 3
[1] 0
3 * 3
[1] 9
3 / 3
[1] 1
(-27) ^ (1/3)
[1] NaN
4 %/% .7
[1] 5
4 %% .3
[1] 0.1
```

R also has some common functions that have straightforward names:

```
sqrt(3)
[1] 1.732051
abs(-3)
[1] 3
exp(1)
[1] 2.718282
log(2.71)
[1] 0.9969486
```

Trigonometric functions also have their part, and `?Trig` can bring up a nice list of these. We show cosine's function call `cos()` for brevity. Note the slight inaccuracy again on the cosine function's output:

```
cos(3.1415)
[1] -1
```

We close this section and this chapter with a brief selection of matrix operations. Scalar operations use the basic arithmetic operators. To perform matrix multiplication, we use %*%:

```
x2
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6

x2 * 3
     [,1] [,2]
[1,]    3   12
[2,]    6   15
[3,]    9   18

x2 + 3
     [,1] [,2]
[1,]    4    7
[2,]    5    8
[3,]    6    9

x2 %*% matrix(c(1, 1), 2)
     [,1]
[1,]    5
[2,]    7
[3,]    9
```

Matrices have a few other fairly common operations that are helpful in linear algebra. We suppose you have some idea about the mathematics behind some of the applications in modeling we cover, and we discuss an appropriate amount of mathematics as needed in the following chapters. Still, this seems a good place to show how the transpose, cross product, and transpose cross product might be coded. We show both the raw code to make the cross product and transpose cross product occur, as well as easier function calls that may be used. This is a relatively common occurrence in R, incidentally. Through packages, quite a few techniques are implemented in fairly clear function calls. Here are the examples:

```
t(x2)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6

t(x2) %*% x2
     [,1] [,2]
[1,]   14   32
[2,]   32   77

crossprod(x2)
     [,1] [,2]
[1,]   14   32
[2,]   32   77
```

```
x2 %*% t(x2)
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45

tcrossprod(x2)
     [,1] [,2] [,3]
[1,]   17   22   27
[2,]   22   29   36
[3,]   27   36   45
```

We end this chapter with some final thoughts . First, as you have just seen, it is common in R for someone else to have done the heavy lifting by making a function that simply creates the desired outcome. Of course, these friendly programmers' work is subjected to only the underlying constraints of R itself as well as the ability to acquire a free GitHub account. Thus, it can be helpful to understand at least some of the base commands and operators that make R work. Second, R runs on computers, and for those who have not yet met computer logic, there are differences due to the hardware structure and (and consequent software implementation choices).

Next, let's focus on understanding implementation nuances as well as quickly getting data in and out of R.

References

- https://mran.microsoft.com/open/
- https://cran.r-project.org/web/packages/checkpoint/vignettes/checkpoint.html

CHAPTER 2

Programming Utilities

Using R to perform more-advanced operations requires creating something that may never have existed. To create new things takes a nuanced understanding of precisely how prebuilt functions work. This chapter discusses how and where to find help and documentation for existing capabilities, how R operates with your computer system and files, and the ins and outs of data input and output. As before, please feel free to pick and choose which parts of this chapter you need. We start with the help files and documentation.

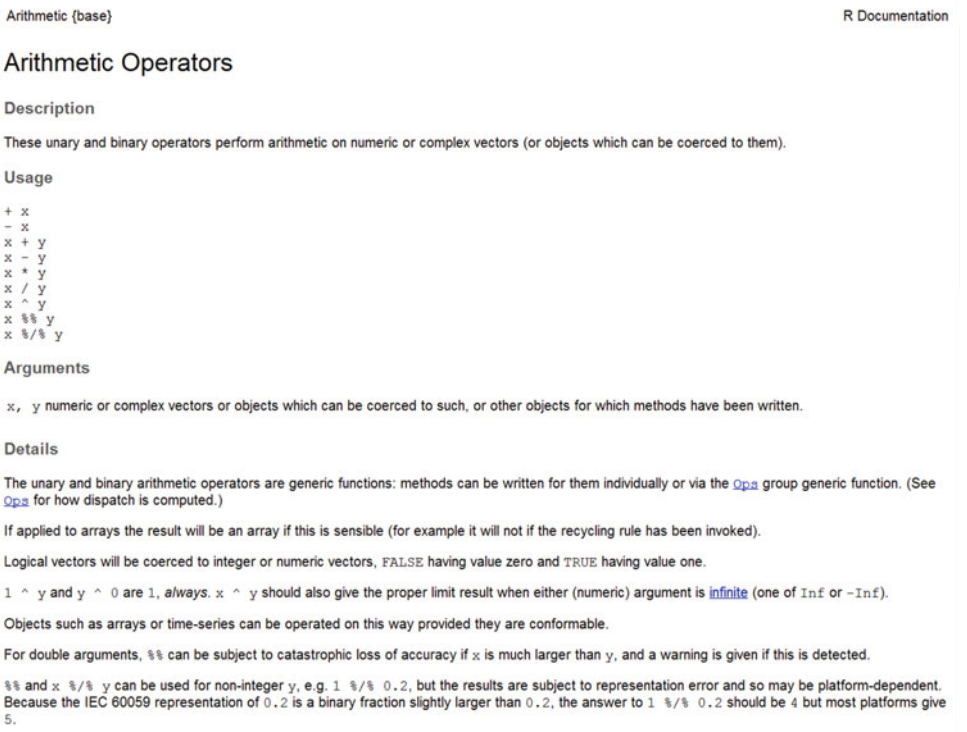
Note Throughout this book, the code in bold is meant to be run. The nonbolded code represents either output results of command lines or code that is intended to inform without necessarily being run.

Help and Documentation

The R community has many resources to help users. From prebuilt functions to whole collections of themed functions in packages, there are many types of support. Both `?` and `help` are useful ways to access information about an object or function. For more common objects, R has not only extensive documentation about specifics such as input (for functions), but also detailed notes on what those inputs are expected to receive. Furthermore, often detailed examples showcase just what can be done. Figure 2-1 shows the output of using these two functions with the addition operator:

```
?'+'
help("+")
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_2



Arithmetic {base} R Documentation

Arithmetic Operators

Description

These unary and binary operators perform arithmetic on numeric or complex vectors (or objects which can be coerced to them).

Usage

```
+ x
- x
x + y
x - y
x * y
x / y
x ^ y
x %% y
x %/% y
```

Arguments

`x, y` numeric or complex vectors or objects which can be coerced to such, or other objects for which methods have been written.

Details

The unary and binary arithmetic operators are generic functions: methods can be written for them individually or via the `Ops` group generic function. (See `Ops` for how dispatch is computed.)

If applied to arrays the result will be an array if this is sensible (for example it will not if the recycling rule has been invoked).

Logical vectors will be coerced to integer or numeric vectors, `FALSE` having value zero and `TRUE` having value one.

`1 ^ y` and `y ^ 0` are `1`, *always*. `x ^ y` should also give the proper limit result when either (numeric) argument is infinite (one of `Inf` or `-Inf`).

Objects such as arrays or time-series can be operated on this way provided they are conformable.

For double arguments, `%%` can be subject to catastrophic loss of accuracy if `x` is much larger than `y`, and a warning is given if this is detected.

`%%` and `x %/% y` can be used for non-integer `y`, e.g. `1 %/% 0.2`, but the results are subject to representation error and so may be platform-dependent. Because the IEC 60059 representation of `0.2` is a binary fraction slightly larger than `0.2`, the answer to `1 %/% 0.2` should be `4` but most platforms give `5`.

Figure 2-1. *Help documentation for arithmetic operators*

Notice at the bottom of Figure 2-1 that the documentation for arithmetic operators is even kind enough to provide specific information about the fact that there can be differences in output depending on the platform. It is important to note that not all functions have this fully complete documentation readily available, but for many of the most common functions and packages, an extraordinary level of information is available. Writing code that reproduces the desired results, independent of platform and environment, is not always possible. Later, when we discuss debugging, it is this sort of advanced knowledge of various functions that can be helpful to know.

Of course, this kind of help is more useful when you already know the object or function you want to use and simply need more details. When seeking the ability to do something entirely new, referring to the manuals can help. One such site is `https://cran.r-project.org/manuals.html` maintained by the *R Development Core Team*.

We turn our attention now to the ways that R can access system files.

System and Files

In a Windows environment, R may be a very effective way to automate file manipulation. Most IT departments are willing to install R, and it has the same file permission privileges you do. From creating files to moving them about and checking dates, R has a variety of functions that are handy for getting information from the system and automating file management. Our observation is that Unix-based systems might have more-elegant ways of handling such scenarios from the command line.

One helpful feature of R accessing the system is that it is possible to discover the current date, time, and time zone of the system on which R is being run. This can help detect new files or can be used to put timestamps into files (more on that later). It should be noted that these are, of course, dependent on the system environment being accurate, so caution may be in order before using these in high-stakes projects.

```
Sys.Date()
[1] "2016-02-13"

Sys.time()
[1] "2016-02-13 16:58:36 CST"

Sys.timezone()
[1] "America/Chicago"
```

The help documentation for these commands makes useful suggestions to increase the accuracy of the output, up to potentially millisecond or microsecond precision.

Let's turn our attention now to a variety file management functions. These all share the format `file.*` and have as their first argument a character vector that should be either a filename for the current working directory or a path and filename. We start with a text file `ch02_text.txt` in our working directory.

The function `file.exists()` takes only a character string input, which is a filename or a path and filename. If it is simply a filename, it checks only the working directory. This working directory is verified by the `getwd()` function. If a check is desired for a file in another directory, this may be done by giving a full file path inside the string. This function returns a logical value of either `TRUE` or `FALSE`. Depending on user permissions for a particular file, you may not get the expected result for a particular file.

```
 getwd()
[1] "C:/Books/Apress_AdvancedR/RFiles"
 file.exists("ch02_text.txt")
[1] TRUE
 file.exists("NOSUCHFILE.FAKE")
[1] FALSE
 file.exists("C:/Books/Apress_AdvancedR/Apress_AdvancedR_Proposal.docx")
[1] TRUE
```

While the preceding function checks for existence, another function tests for whether we have access to a file. The function `file.access` takes similar input, except it also takes a second argument. To test for existence, the second argument is `0`. To test for executable *permissions*, use `1`; and to test for writing permission, use `2`. If you want to test your ability to read a file, use `4`. This function returns an integer vector of `0` to indicate that permissions are given, and `-1` to indicate permissions are not given. Notice that the default for the function is to test for simple existence. Examples of `file.access` are shown here:

```
file.access("ch02_text.txt")
ch02_text.txt
           0
file.access("ch02_text.txt", mode = 0)
ch02_text.txt
           0
file.access("ch02_text.txt", mode = 1)
ch02_text.txt
          -1
```

```
file.access("ch02_text.txt", mode = 2)
ch02_text.txt
           0
file.access("ch02_text.txt", mode = 4)
ch02_text.txt
           0
```

We next turn our attention to more detailed information about when a file was modified, changed, or accessed with the `file.info` function. This function takes in character strings as well. The output gives information about the file size; whether it is a directory; a file permissions integer in read, write, and execute order; the last modified time; the last change time; the last accessed time; and finally, whether the file is executable:

```
file.info("ch02_text.txt", "chapter01.R")
             size isdir mode               mtime               ctime               atime exe
ch02_text.txt   31 FALSE  666 2016-02-13 17:00:16 2016-02-13 16:59:57 2016-02-13 16:59:57  no
chapter01.R   7983 FALSE  666 2016-01-01 02:53:17 2016-01-05 12:26:39 2016-01-01 02:53:17  no
```

Notice that you can edit the modified time through the `sys.setFileTime` function. This can be helpful on occasion, although the precise accuracy and precision are dependent on the environment. Here's an example:

```
newTime<-Sys.time()-20
newTime
[1] "2016-02-13 20:25:53 CST"

file.info("ch02_text.txt")
             size isdir mode               mtime               ctime               atime exe
ch02_text.txt   31 FALSE  666 2016-02-13 17:00:16 2016-02-13 16:59:57 2016-02-13 16:59:57  no

Sys.setFileTime("ch02_text.txt", newTime)
file.info("ch02_text.txt")
             size isdir mode               mtime               ctime               atime exe
ch02_text.txt   31 FALSE  666 2016-02-13 20:25:53 2016-02-13 16:59:57 2016-02-13 16:59:57  no
```

Turning our attention to creation and removal of files, the functions `file.create` and `file.remove` do precisely what you would hope. These do return logically `TRUE` or `FALSE` and can even give more details:

```
file.create("ch02_created.docx", showWarnings = TRUE)
[1] TRUE
file.remove("ch02_created.docx")
[1] TRUE
file.remove("NOSUCHFILE.FAKE")
[1] FALSE
Warning message:
In file.remove("NOSUCHFILE.FAKE") :
 cannot remove file 'NOSUCHFILE.FAKE', reason 'No such file or directory'
```

Files may also be copied and renamed. The function `file.copy` can be given overwrite permission, and could even be set up to copy entire folders and subfolders with the `recursive=TRUE` option. Furthermore, it has options to copy over mode or file permissions as well as to copy the file date data (or, of course, letting the copy have a new modified date). The following code example shows how that might all work:

```
Sys.time()
[1] "2016-02-13 21:05:29 CST"
file.copy("ch02_text.txt", "ch02_copy.txt", overwrite = TRUE, recursive = FALSE, copy.mode =
TRUE, copy.date = TRUE)
[1] TRUE
file.info("ch02_copy.txt")
              size isdir mode               mtime               ctime               atime exe
ch02_copy.txt   31 FALSE  666 2016-02-13 20:25:53 2016-02-13 21:05:30 2016-02-13 21:05:30  no
file.rename("ch02_copy.txt", "ch02.txt")
[1] TRUE
```

The `file.append` function joins two files together. This can work well for some files types. Used naively, there can be unfortunate consequences. While the code to perform this follows, pay careful attention to Figure 2-2 and Figure 2-3. In the case of the text files, the process worked well enough. In the case of Microsoft PowerPoint files, not so much. Keep in mind that the PowerPoint files are less about the files themselves (neither author imagines using R for such files genuinely), and more about the fact that the nuances of file manipulation, in general, deserve treatment with due caution. Notice that R believes both operations are successful:

```
file.append("ch02_text.txt", "ch02.txt")
[1] TRUE
file.append("ch02_pp.pptx", "ch02_pp2.pptx")
[1] TRUE
```

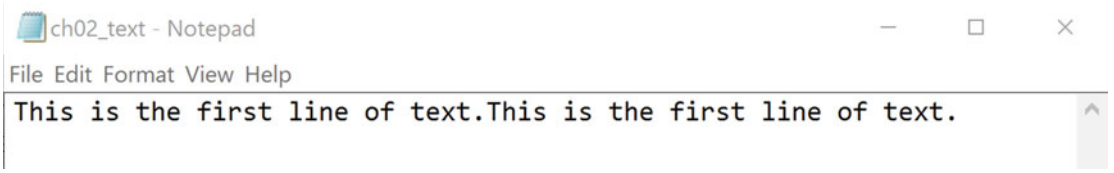


Figure 2-2. *The first operation was successful according to R and worked upon opening the file*

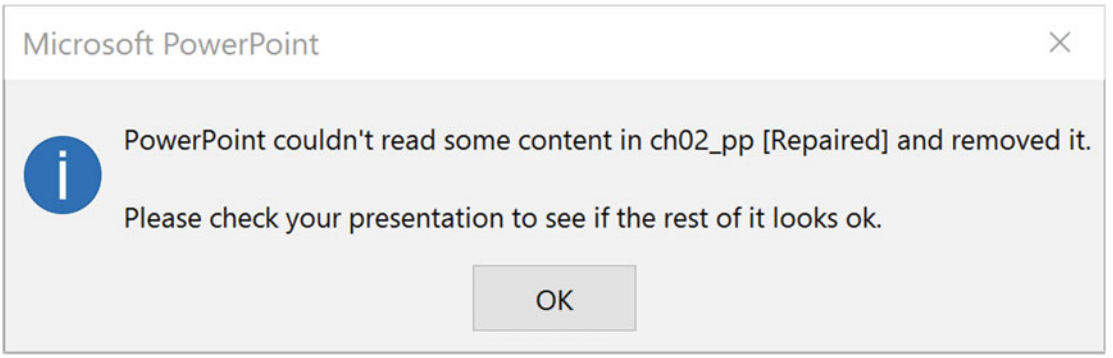


Figure 2-3. *The second operation was successful according to R yet did not work upon opening the file*

You can not only manipulate files with R, but also create directories, or file folders, as well. The function is `dir.create`, which behaves as you might now expect. We show an example of the function here, and the resulting director in Figure 2-4:

```
dir.create("folder1")
```

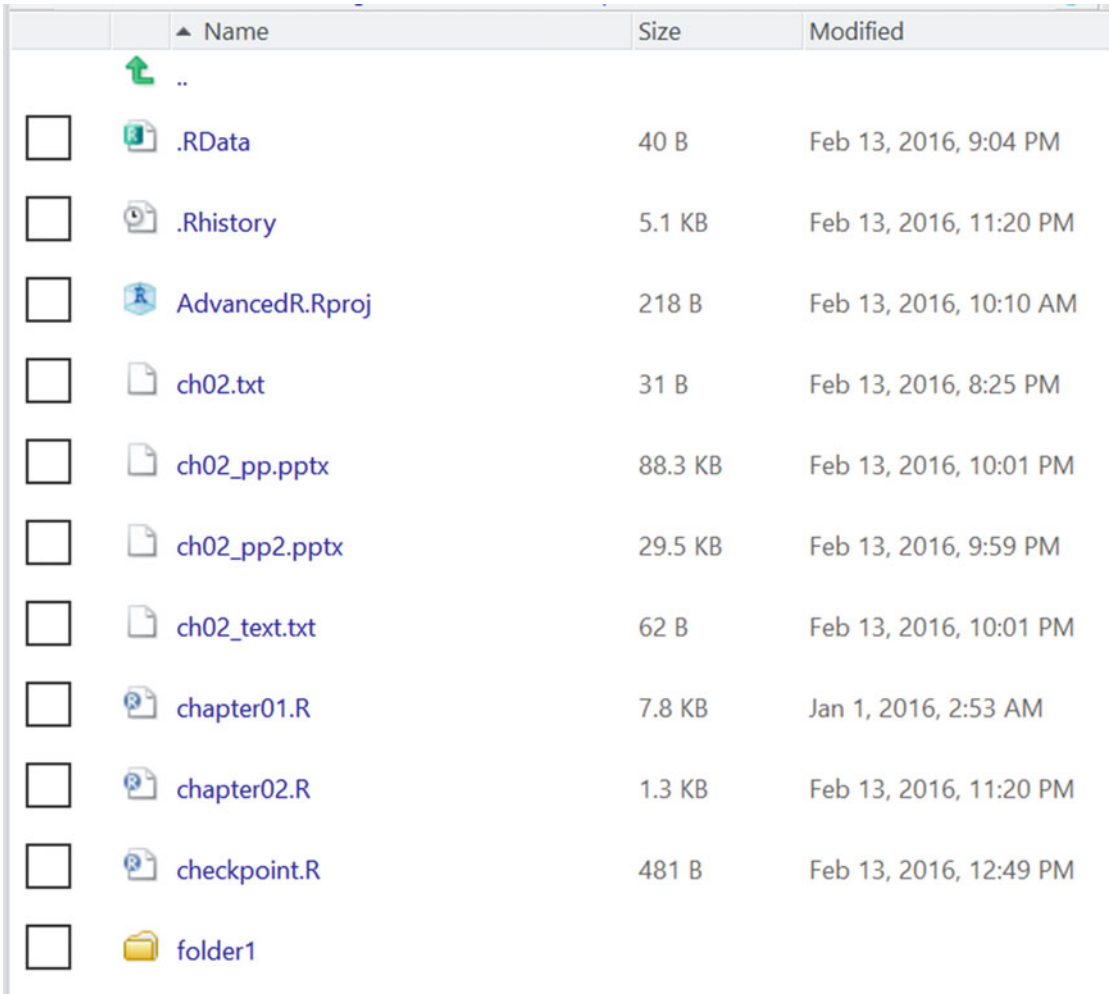


Figure 2-4. *The created folder1 directory*

These commands work on any files that a user has permission to access and manipulate. One of the authors uses these commands to move data from various shared drives owned by different departments to eventually post result files on a website. Once you understand loops and functions from future chapters, you'll be able to automate most file management.

Input

Getting new data into R becomes the next challenge. Data sets tend to be quite large, although effective techniques may be used on smaller sets. Text files with tab or comma separation are the most straightforward to import into R. Next, common data file types include Microsoft Excel, SPSS, SAS, and Stata. More generally, it is fairly safe to say that there likely exists an R package that can handle the type of file import you want. Even PDF and Microsoft Word files may be input should the need arise (text analytics from word clouds to more predictive applications come to mind). For most of these records, the input process is similar, so there is perhaps less of a need to be exhaustive and more of a need to set up sound principles. Be sure to visit the Apress website for this text to download the code packets for this book. We use files in the chapter 02 folder in our next examples; the `Counties in Illinois` files (All Counties in Illinois, 2016) and the `rscfp2013` files (DADS, 2016) are used.

We start with a function in R, `read.table()`, which can take in several of the more basic file types and read them in as a data frame. As with most input functions, this has several options, not all of which are required for any particular circumstance. Depending on the type of data read into R, it may take more than one try to successfully read in the data in a way convenient to use and manipulate. The `View` function can be of help in this case, with the output shown in Figure 2-5.

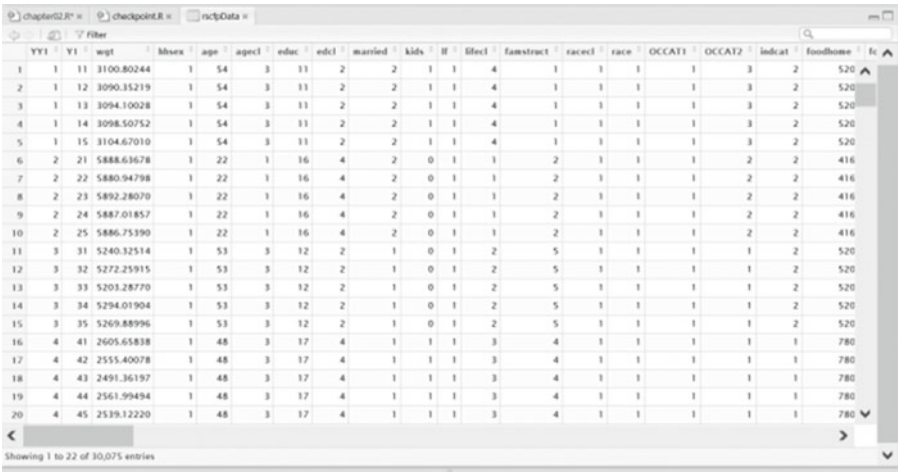
```
countiesILCSV<-read.table("Ch02/Counties_in_Illinois.csv", header = TRUE, sep = ",")
View(countiesILCSV)
```

	county_name	total_population	median_income	less_than_high_school	high_school	some_college	bachelors_or_higher	inc_greater_than_ave	greate
1	Adams County, Illinois	67030	43824	0.06938496	0.3663752	0.3155484	0.24869141	1	
2	Alexander County, Illinois	8449	28833	0.17821309	0.3999044	0.3251314	0.09675108	1	
3	Bond County, Illinois	17904	51946	0.08590590	0.3386204	0.3054077	0.27006600	0	
4	Boone County, Illinois	53567	61210	0.12358696	0.3577536	0.2985507	0.22010870	0	
5	Brown County, Illinois	6897	38696	0.24846045	0.3012790	0.3270962	0.12316438	1	
6	Bureau County, Illinois	35083	45692	0.09044028	0.3910391	0.3542198	0.16430076	1	

Figure 2-5. *The output of the View() function*

We use three packages—`Hmisc` (Harrell, 2016), `xlsx` (Dragulescu, 2014), and `foreign` (R Core Team, 2015)—to showcase input from various file types. One observation is that file types may, in fact, be quite large. With over 30,000 entries, as shown in Figure 2-6, `read.dta` quickly and handily imports Stata files.

```
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(Hmisc)
library(foreign)
library(xlsx)
rscfpData <- read.dta("Ch02/rscfp2013.dta")
View(rscfpData)
```



	YY1	Y1	wgt	hhsex	age	agecl	educ	edcl	married	kids	lf	lifecl	famstruct	racecl	race	OCCAT1	OCCAT2	indcat	foodhome	fc
1	1	11	3100.80244	1	54	3	11	2	2	1	1	4	1	1	1	1	3	2	520	
2	1	12	3090.35219	1	54	3	11	2	2	1	1	4	1	1	1	1	3	2	520	
3	1	13	3094.10028	1	54	3	11	2	2	1	1	4	1	1	1	1	3	2	520	
4	1	14	3098.50752	1	54	3	11	2	2	1	1	4	1	1	1	1	3	2	520	
5	1	15	3104.67010	1	54	3	11	2	2	1	1	4	1	1	1	1	3	2	520	
6	2	21	5888.63678	1	22	1	16	4	2	0	1	1	2	1	1	1	2	2	416	
7	2	22	5880.94798	1	22	1	16	4	2	0	1	1	2	1	1	1	2	2	416	
8	2	23	5892.28070	1	22	1	16	4	2	0	1	1	2	1	1	1	2	2	416	
9	2	24	5887.01857	1	22	1	16	4	2	0	1	1	2	1	1	1	2	2	416	
10	2	25	5886.75390	1	22	1	16	4	2	0	1	1	2	1	1	1	2	2	416	
11	3	31	5240.32514	1	53	3	12	2	1	0	1	2	5	1	1	1	1	2	520	
12	3	32	5272.25915	1	53	3	12	2	1	0	1	2	5	1	1	1	1	2	520	
13	3	33	5203.28770	1	53	3	12	2	1	0	1	2	5	1	1	1	1	2	520	
14	3	34	5294.01904	1	53	3	12	2	1	0	1	2	5	1	1	1	1	2	520	
15	3	35	5269.88996	1	53	3	12	2	1	0	1	2	5	1	1	1	1	2	520	
16	4	41	2605.65838	1	48	3	17	4	1	1	1	3	4	1	1	1	1	1	780	
17	4	42	2555.40078	1	48	3	17	4	1	1	1	3	4	1	1	1	1	1	780	
18	4	43	2491.36197	1	48	3	17	4	1	1	1	3	4	1	1	1	1	1	780	
19	4	44	2561.99494	1	48	3	17	4	1	1	1	3	4	1	1	1	1	1	780	
20	4	45	2539.12220	1	48	3	17	4	1	1	1	3	4	1	1	1	1	1	780	

Showing 1 to 22 of 30,075 entries

Figure 2-6. *A larger Stata .dta file successfully imported into R with thousands of entries*

We can also import SPSS files through the `spss.get` function. Even if there are warnings, it can often be the case that data is still successfully imported. If the data is not imported successfully, search the warning message for specifics. In this case, it seems from our header view that all is well. For brevity's sake, we truncated part of the header output:

```
countiesILSPSS <- spss.get("Ch02/Counties_in_Illinois.sav")
Warning message:
In read.spss(file, use.value.labels = use.value.labels, to.data.frame = to.data.frame,  :
  Ch02/Counties_in_Illinois.sav: Unrecognized record type 7, subtype 18 encountered in
system file

head(countiesILSPSS)
                    county.name total.population median.income
1 Adams County, Illinois                   67030         43824
2 Alexander County, Illinois                8449         28833
3 Bond County, Illinois                    17904         51946
4 Boone County, Illinois                   53567         61210
5 Brown County, Illinois                    6897         38696
6 Bureau County, Illinois                  35083         45692
  less.than.high.school high.school some.college bachelors.or.higher
1            0.06938496   0.3663752    0.3155484          0.24869141
2            0.17821309   0.3999044    0.3251314          0.09675108
3            0.08590590   0.3386204    0.3054077          0.27006600
4            0.12358696   0.3577536    0.2985507          0.22010870
5            0.24846045   0.3012790    0.3270962          0.12316438
6            0.09044028   0.3910391    0.3542198          0.16430076
```

We'll do one final example with Excel, which uses our last package, `xlsx`. This package has `rJava` (Urbanek, 2016) as a dependency, and that is relevant depending on which R version you use. Your mostly fearless authors stick to 64-bit as often as possible, and this requires a 64-bit version of Java installed. R was liberal with its complaints when this had not been done. Notice that sheet names can be specifically called, making the function `read.xlsx` handy for extracting specific pieces of data.

```
countiesILExcel <- read.xlsx("Ch02/Counties_in_Illinois.xlsx", sheetName =
"Counties_in_Illinois")
```

These three packages, along with R's more inherent ability to read in tabular data stored in text files with various delimiters, allow for easy enough input of most data that might be presorted or collected. It is not difficult to direct R to look directly online for files either, so that one researcher may update records and those results can be readily percolated to others. Later, as part of other examples, we have some files that are downloaded live from the Internet. For now, we turn our attention to output.

Output

Output comes in many forms. Perhaps because of collaboration with other researchers or partners, accommodating one of the other software systems is needed. Much like the preceding section on input, R can readily output to several file types. Of more interest is setting up R to send specific console output to certain files. This allows one machine to view the results of an analysis run on another computer. In this section, we demonstrate a couple of outputs of data to SPSS, Stata, and Excel. Then we work with console outputs. We ask you to keep in mind that there are many other ways and types of files to create, and as part of larger examples, we demonstrate several types including various document files and graphics.

To output files to Excel, SPSS, or Stata, simply use the correct invocation of either the `xlsx` or `foreign` packages. As shown in Figure 2-7, R creates the output handily.

```
write.xlsx(countiesILExcel, "Ch02/Output1.xlsx")

write.foreign(countiesILExcel, "Ch02/Output2.txt", "Ch02/Output2.sps",   package="SPSS")
Warning message:
In writeForeignSPSS(df = list(county_name = c(1L, 2L, 3L, 4L, 5L,  :
  some variable names were abbreviated

write.dta(countiesILExcel, "Ch02/Output3.dta")
Warning message:
In write.dta(countiesILExcel, "Ch02/Output3.dta") :
  abbreviating variable names
```

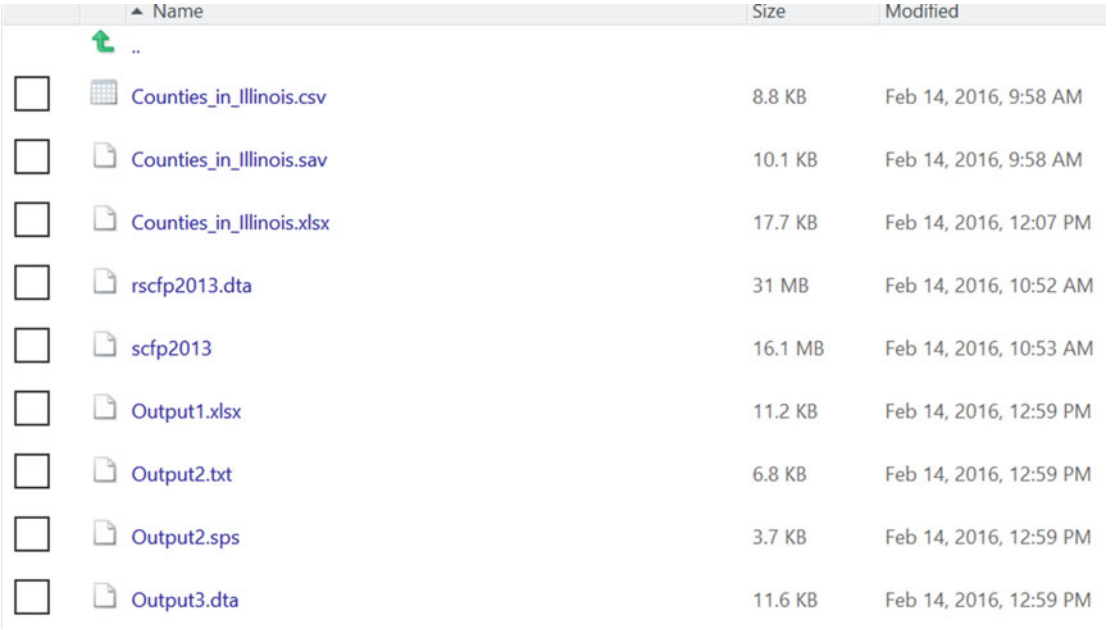


Figure 2-7. *Output in Excel, SPSS, and Stata file formats is created from R*

The `sink` function takes console output and directs it to a file. Thus, the results of an R process may be stored for later observation or saved to a shared drive for perusal by others. The console sends only output to the file, not the input. Look carefully at the difference between the code that follows and the screenshot of `Output4.txt` shown in Figure 2-8:

```
sink("Ch02/Output4.txt", append = TRUE, split = TRUE)
x <- 10
xSquared <- x^2
x
[1] 10
xSquared
[1] 100
unlink("Ch02/Output4.txt")
```

Figure 2-8. *The output of the sink() function to a text file*

We turn our attention away from the output for a little while, knowing that we'll revisit this topic in several more chapters. Output of various types are necessary, and they tend to depend on objectives and circumstances.

The next chapter provides tools for quickly repeating similar operations again and again as well as handling course corrections based on the environment. Those techniques combine with these methods to quite handily automate file management on a relatively large scale.

References

References are given once and not repeated for packages. Data files found online are cited when used. Our goal is to give credit where it is due, without overloading the text.

CHAPTER 3

Programming Automation

It has been said that performing the same action and expecting a different result is a definition of *insanity*. While a brief search of Oxford's dictionary does not turn up that definition, we have a similar premise: to prevent insanity, leave it to your computer to perform the same action repeatedly, and expect the same results!

The goal of this chapter is for you to begin to build automation into your code. Part of the power of code is that it cheerfully performs the same action as often as required without stumbling or tiring. The other useful feature of code is the capability to stitch logic into the flow of the programming. Humans, at our best, naturally use such logic. For example, if father's keys are on the hook, he must be home. Otherwise, father must still be out and about. Here's another example: if *p* is less than *alpha*, reject the null hypothesis; otherwise, do not reject the null hypothesis.

Because such automation allows an enormous number of repeats, care must be given to efficiency. How long does it take the code to run? Could the code be written differently to make the operations occur faster? In programming, as in life itself perhaps, we often have fewer perfect answers and more trade-offs between choices and consequences. We live in a brave new world in which new and more powerful hardware often is cheaper than the human cost required to squeeze out a bit more efficiency. On the other hand, our brave new world also has data sets with billions of entries; shaving a millisecond off each calculation could save hundreds of hours of compute time. We do our best to take a balanced approach, demonstrating some of the easier-to-understand constructs first and then presenting faster methods.

As we look at the first of these automation methods, we remind you that coding is as much an art as it is a science. The cleaner, more readable code may not be the fastest. The only essential definition of *fast* is *fast enough*. If a particular type of code makes more sense to you and thus makes you more likely to remember and use it, that may be good enough. Of course, if it is not, a bit of research is in order to uncover quicker alternatives.

Loops

Loops repeat the code inside them. The concern is to avoid getting into a case of an infinite loop—one that lasts forever. Most loops have a built-in means to attempt a stop (not that those always work as planned). In the next section, we discuss ways to exit a loop manually. Another concern with loops has to do with the fact that they are functions. Although we have used that word already, we are postponing a frank discussion of functions until the next chapter. For now, just keep in mind that anything created inside a function may disappear when that function ends. In the case of loops, this happens when they stop repeating. So, if you want to hold onto results, you need to build the container for those results outside the loop. Again, we go deeper into specifics in the next chapter.

The `for` loop is the first (and perhaps most controversial in R) automator we discuss. In many computer languages, `for()` is considered rather fast, but not so in R. Nevertheless, this function is easy to understand and use. Human-readable code is not something to eschew needlessly. However, there is usually more than one way to do things, and it would be silly to ignore those. We use the function `proc.time()` in our code to measure different times; we will calculate the runtime by finding the difference between the stop and start time. Take a look at the following code and output:

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_3

```
x <- seq(1,100000,1)
head(x)
[1] 1 2 3 4 5 6
forTime<- proc.time()
xCube <- NULL
for (i in 1:length(x) ) {
+   xCube[i] = x[i]^3
+ }
forTime <- proc.time()-forTime
head(xCube)
[1]   1   8  27  64 125 216
forTime
   user  system elapsed
  11.04    0.17   11.25
```

You can see that the numbers 1 through 100,000 are in the variable x (the first six display by calling the head() function). Next, we create a variable to hold the cubes of all those variables. We also store the current time in a variable so that we can time the operation. Now comes the magic of the for() function! This function takes an index, which is often called i, and a range that the index operates over. We start i at 1, and the loop runs. At the end of the loop, i is automatically incremented by 1, and the loop is repeated. The loop repeats until it reaches the last part of the range. Notice that we do not choose a hard limit; rather, we make our loop adaptable based on the length of x. You can see that the loop does, in fact, result in each term of x being cubed. It takes 11.25 seconds on our system.

This time, though, can be improved. Notice that although we created a variable xCube, it is a null variable. Each iteration of our for loop not only needs to cube x, but also has to increase the size of the xCube variable by 1. This turns out to not be a trivial task. If we were to create xCube to be already the size we need, performance would significantly improve. We first remove xCube, so that we are not being deceived by any prior data:

```
rm(xCube)
forTime2<- proc.time()
xCube <- vector(mode = "numeric", length = length(x))
for (i in 1:length(x) ) {
+   xCube[i] = x[i]^3
+ }
forTime2 <- proc.time()-forTime2
head(xCube)
[1]   1   8  27  64 125 216
forTime2
   user  system elapsed
   0.15    0.00    0.15
```

Notice that now our time is only 0.15 seconds—much improved! Of note is that even if we had created a dummy xCube that wasn't quite long enough, or if we made one that was too long and needed some null entries deleted off the end, this would still be faster. Thus, even when it may not be possible or convenient to know the size of the variable needed before running the loop, it may be beneficial to create a container that is likely large enough to store the data in first.

Now, for this particular example, a for loop is the wrong way to go. R has powerful and fast underlying code for some of its simplest functions. Notice that there is almost no elapsed time at all (of course, there is some, but depending on the operating system, some differences are not noticeable). Also, notice that the head() function can take negative numbers, to count backward from the end, to show the following:

```
xCube <- NULL
vTime <- proc.time()
xCube <- x^3
vTime <- proc.time()-vTime
head(xCube, -99994)
[1]   1   8  27  64 125 216
vTime
   user  system elapsed
      0       0       0
```

Other types of loops exist. The `for` loop is best suited to running a process a certain number of times, and, when it gets to the end, stopping. The `while` loop, on the other hand, is best suited to running a process an uncertain number of times until a stop condition happens. Often, it is possible to gain the same results with different types of loops (or as you have just seen, without a loop at all). All the same, each type of loop tends to have a more natural use under certain conditions. As an exercise on your own, after studying the `while` loop example we give, see if you can duplicate the previous cubing results.

Perhaps an example near and dear to the heart of researchers everywhere is simulation. After all, if we can simulate data, that may be the first step to understanding what our real-world results may be. Statistics are baked into R, and the function `rnorm()` takes up to three inputs. The first controls the number of elements we want to return for our sample. The second and third control the population-level mean and standard deviation from which the sample is randomly selected. Let us a look at what `rnorm` gives us:

```
for (i in 1:5 ) {
+   x <- rnorm(5,4,2)
+   print(x)
+   print(paste0("Xbar: ",mean(x)))
+   print(paste0("StdDev: ",sd(x)))
+ }
[1] 2.0287126 4.5664489 0.1310789 3.6347667 4.2465319
[1] "Xbar: 2.92150781627209"
[1] "StdDev: 1.84077667180739"
[1] 5.806652 2.707348 4.673238 4.646211 6.496605
[1] "Xbar: 4.86601066066082"
[1] "StdDev: 1.43952629897616"
[1] 1.6476375 0.8618630 0.2519337 0.6902068 1.0062915
[1] "Xbar: 0.8915864810408"
[1] "StdDev: 0.508764053558126"
[1] 5.162450 3.513966 6.503733 3.736576 5.503810
[1] "Xbar: 4.88410690747199"
[1] "StdDev: 1.25287757407054"
[1] 4.719578 1.197946 2.751889 4.451728 2.487819
[1] "Xbar: 3.12179207747434"
[1] "StdDev: 1.46300892878659"
```

What we find are 5 samples randomly pulled from the same population with a mean of 4 and a standard deviation of 2. However, notice that each sample has means ranging from 0.89 to 4.88. Suppose we need a pseudorandom sample such as this, but want to demonstrate to our students how random sampling-point estimators require caution. Let us say we want to find a random sample pulled from our population, but whose sample mean is in fact 10 or higher. Well, a `while` loop might be just the answer:

```
y <- 3
while(mean(y)< 10){y <- rnorm(5, mean = 4.0, sd = 2)}
```

We allowed this code to run for about a minute before we put a stop to it. The reason, of course, is that this type of search could take a while, and runs dangerously close to being an infinite loop. Normal populations whose means are 4 with standard deviations of 2 tend to have few elements that are 10 or higher. Randomly sampling enough of those elements that the sample mean is 10 or higher starts to be unlikely. We set y to 3 just so that the `while` code would run at least once. Here we repeat our loop with something more reasonable—a mean of 6:

```
y <- 3
while(mean(y)< 6){y <- rnorm(5, mean = 4.0, sd = 2)}
mean(y)
[1] 6.188268
y
[1] 11.126657  4.807775  3.703673  3.966180  7.337056
```

Notice that y is now a rather strange sample, compared to what we know the population to be. This is, of course, an argument for larger sample sizes, but it also gives us data that both fits our facts and tests our assumptions. Counterexamples and odd data can be quite useful to stress test code or models.

The last loop we will look at for the moment is `repeat()` which can be different from `while` and `for` loops. In particular, `repeat`, unlike `while` or `for`, does not take any arguments that would ever have a chance to stop it. This function keeps repeating over and over. Of course, there has to be a way to stop this. Otherwise, we are in trouble. We discuss such techniques in the next section, "Flow Control." However, first, some observations about loops.

Loops are meant to perform a task as many times as required. In R, we control these tasks in three ways. If the task should be done to a certain count, a `for` loop is likely best. An integer iterator controls the start and stop of the `for` loop.If the task is one that we want to be done until a certain condition occurs, a `while` loop may be best. In particular, a `while` loop tests the condition that determines whether the loop runs, before the loop runs. This is critical! If the condition is already met, the `while` loop never runs. In the short examples you've seen so far, this may not seem possible or apparent, and that is okay. Our goal here is to introduce these in a way that easily brings them to mind when you are facing a new problem. All the same, it bears repeating: a `while` loop first checks whether it needs to run, and only then will it run once, and check again, until the check stop condition happens. The last loop we discuss, `repeat()`, is different. It simply runs. Now, we have to stop it somehow, and we do that manually in the next section. The point is, `repeat` never checks whether it needs to run; it just runs. So if you want to run a process at least once, `repeat` may be the function you seek.

We hold off on giving an example of a `repeat` for now, mostly to avoid infinite loops. We'll show an example soon. First, we need a way to control the flow of our code.

Flow Control

Code flow control happens in several ways, but our interests lie in four of them (and just three). This control works by performing tests to decide whether one action or another should be taken, or by modifying the behavior of the loops you have just seen. We present each of the commands in turn, and, at the end of this section, we'll get back to that function mentioned in the prior section.

If/else statements are a standard part of logic, and they make up a standard part of programming as well. In fact, if/else is used so often that it comes in two flavors. When the *else* part is to carry on as usual, it is standard practice not to include that part in the code at all.

Let us take that sample of values for which we forced a sample mean greater than 6. Say we want to see a box plot and whisker plot for that if we are successful, but otherwise want to see a histogram. We might start off with the following broken code, along with its Figure 3-1 output:

```
if(y>6){
+   boxplot(y)
+ }
Warning message:
In if (y > 6) { :
  the condition has length > 1 and only the first element will be used

else{
Error: unexpected 'else' in "else"
   histogram(y)
}
Error: unexpected '}' in "}"
```

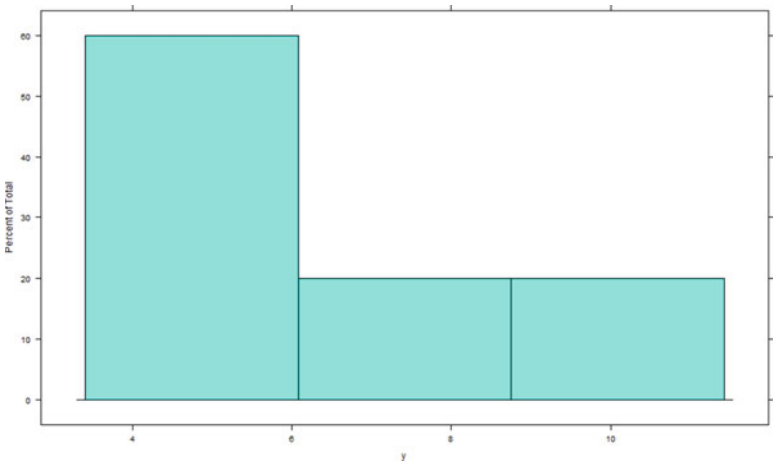


Figure 3-1. *The else output of both the preceding and the following code*

We broke this code for two reasons. The first is to demonstrate that although flow control can be a useful feature, it is only as good as your logic. We meant to test for `mean(y) > 6`, and instead have tested something else. R is helpful enough to warn us that all may not be well. All the same, we should be cautious. The second reason is to note that this if/else statement is coded incorrectly. The `else` portion always runs; R does its best to warn us again that the `else` was not expected, and thus we should take heed. This modified set of code creates two images, shown in Figure 3-1 and Figure 3-2. We show the code as follows, along with Figure 3-2, which is the output of the successfully executed `if` portion:

```
if(mean(y)>6){
+   boxplot(y)
+ }

else{
Error: unexpected 'else' in "else"
   histogram(y)
}
Error: unexpected '}' in "}"
```

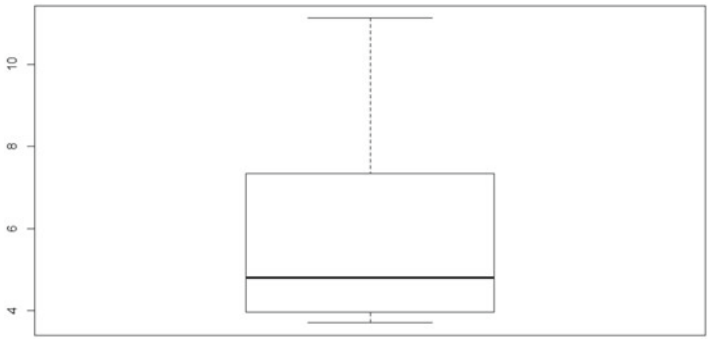


Figure 3-2. *The if output of the preceding code, which has been fixed on the control level but not the else portion*

This is interesting for three reasons. First, as you can see in our first broken example, `if` is fully able to be a stand-alone piece of flow control. The control portion evaluates, and if `true`, the code is run, and if `false`, the code is not run. Second, although less usefully, `else` is also a stand-alone piece of code (which controls nothing when used solo). Third, it is a good object lesson that both the logic of the flow control and the structure of the code must be set up correctly to work properly. Next, we show the code as we originally envisioned it. It outputs only the box plot and whisker plot of Figure 3-2. Notice that the `else` needs to be on the same line as the closing } of the `if`:

```
if(mean(y)>6){
+   boxplot(y)
+ } else{
+   histogram(y)
+ }
```

The other two functions we use are somewhat similar to each other. We use `next` to skip one pass through a loop, and we `break` to end the loop entirely. Both of these keep the code running; they just control what happens inside the loop in which they are called. Take a look at the following code to see what happens:

```
i <- 0
while(i < 5){
+   i <- i + 1
+   if(i ==2) {next()}
+   print(i)
+ }
[1] 1
[1] 3
[1] 4
[1] 5
```

It is important to recognize that the location of `next` is significant.. We could order those three lines of code inside the `while` function in six ways. At least one way creates an infinite loop, and at least one way has no noticeable influence whatsoever. Of course, as you just saw, there is also a way that makes it skip the number `2`. This also demonstrates that `while` can be a lot like a `for` function.

We close this section with a real look at the `repeat` function. This function is ideal when it is unclear how many times an action may need to be repeated. Again, `repeat` is similar to `while`, with the exception that the code runs at least once. It is also easier perhaps to have more than one exit criteria. Granted, `break` can be used in `while` as well to create more than one exit rule. However, for human-readable code, it may be easier to have multiple `break` statements near each other. In the following code, we also recycle our iterator `i` so that it is easy to see that it takes `377` attempts to find our randomly drawn value that is over three standard deviations away from mean:

```
z <- NULL
i <-1
repeat{
+   z <- rnorm(1,4,2)
+   i <- i + 1
+   if(z > 10){break}
+ }
z
[1] 10.69076
i
[1] 378
```

*apply Family of Functions

Our last set of functions for this chapter are the `*apply` functions. Generally speaking, these functions take two primary types of input. One is an input that has several elements, and the second is a function applied to the elements in turn. The various flavors handle different use cases, and we also introduce some error checking where possible.

We start with `lapply`, which takes a list input and applies the function given to each element. As its prefix suggests, this function returns a list with the results of the application of the entered function:

```
xL <- 1:5
xL
[1] 1 2 3 4 5

## returns a list
lapply(xL, is.integer)
[[1]]
[1] TRUE

[[2]]
[1] TRUE

[[3]]
[1] TRUE

[[4]]
[1] TRUE

[[5]]
[1] TRUE
```

As you can see, this is fairly messy. To simplify, we use almost the same function call, but with the `s` prefix (for *simple*) as follows:

```
sapply(xL, is.integer)
[1] TRUE TRUE TRUE TRUE TRUE
```

Although both of these are helpful enough in their own right, they depend on correct input. Of course, our code depends on such correct input (the phrase *garbage in, garbage out* comes to mind). Nevertheless, we can signal to R the type of results we are expecting with the `vapply` function. It takes a third input, which is set to the type of output we are expecting. What happens when the function does not return the correct type of value? Here we run two lines of code, telling our function to expect `logical` level inputs because by default `NA` is a `logical`. In our second attempt, R still expects the same type, but gets a `double` instead:

```
vapply(xL, is.integer, FUN.VALUE = NA)
[1] TRUE TRUE TRUE TRUE TRUE

vapply(xL, mean, FUN.VALUE = NA)
Error in vapply(xL, mean, FUN.VALUE = NA) :
  values must be type 'logical',
 but FUN(X[[1]]) result is type 'double'
```

We run a similar set of code, this time telling R to expect each result to be a vector of length `2`. Instead, it gets a vector of length `1`, which is the correct type, yet wrong length:

```
vapply(xL, is.integer, FUN.VALUE = c(NA, NA))
Error in vapply(xL, is.integer, FUN.VALUE = c(NA, NA)) :
  values must be length 2,
 but FUN(X[[1]]) result is length 1
```

Let us take a look at a more interesting list. Notice that we have both integers and characters in this list `yL`:

```
yL <- list(  a = 1:5,  b = c("6", "7", "8"),  c = c(9:10))
yL
$a
[1] 1 2 3 4 5

$b
[1] "6" "7" "8"

$c
[1]  9 10
yL$b[2]
[1] "7"
is.character(yL$b[2])
[1] TRUE
is.integer(yL$b[2])
[1] FALSE
```

Suppose we are interested in summary statistics information for each element in our list. We could certainly apply the `summary()` function to each element, and R would cheerfully do so, and not even throw an error at us!

```
lapply(yL, summary)
$a
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
      1       2       3       3       4       5

$b
   Length     Class      Mode
        3 character character

$c
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   9.00    9.25    9.50    9.50    9.75   10.00
```

Notice, however, that there are some issues. This is an easy sort of data-collection typo to have, depending on how information was collected in the field and coded. Luckily, if we use a better `*apply` function and give R a heads-up on what to expect, we can potentially save ourselves some grief later. Building some form of data validation into code can be helpful. Of course, this does not prevent all issues, by any means; it simply tends to make life a bit safer.

```
vapply(yL, summary, c(Min. = 0, `1st Qu.` = 0, Median = 0, Mean = 0, `3rd Qu.` = 0, Max. =
0))
Error in vapply(yL, summary, c(Min. = 0, `1st Qu.` = 0, Median = 0, Mean = 0,  :
  values must be length 6,
 but FUN(X[[2]]) result is length 3
```

The `mtcars` data frame comes from a 1974 magazine. We are interested in the miles per gallon and the number of cylinders. Using this data set is a useful way to understand various functions in R, as it is readily accessible in R. Here we take a look at the first six entries to see and better understand our raw data:

```
head(mtcars[, c("mpg", "cyl")])
                   mpg cyl
Mazda RX4         21.0   6
Mazda RX4 Wag     21.0   6
Datsun 710        22.8   4
Hornet 4 Drive    21.4   6
Hornet Sportabout 18.7   8
Valiant           18.1   6
```

We run `tapply` on the `mtcars` data frame, and we factor by cylinders. In Figure 3-3, we show what this looks like as a bar plot.

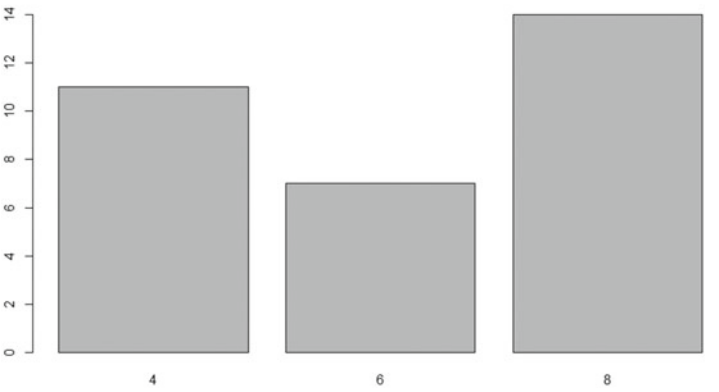


Figure 3-3. *mtcars data frame broken out by number of cylinders*

Notice that in our function call this time, we are explicitly stating each of the inputs by name. That is to say, `X`, `INDEX`, and `FUN` are the original names of the data, the factor input, and the function that is applied, respectively. We also show almost the same code with the `simplify = FALSE` option, which returns a list:

```
tapply(
+   X = mtcars$mpg,
+   INDEX = mtcars$cyl,
+   FUN = mean)
       4        6        8
26.66364 19.74286 15.10000

tapply(
+   X = mtcars$mpg,
+   INDEX = mtcars$cyl,
+   FUN = mean,
+   simplify = FALSE)
$`4`
[1] 26.66364

$`6`
[1] 19.74286

$`8`
[1] 15.1
```

When our data structure is a data frame, matrix, or array, the `apply()` function is often a good choice. Because these tend to have more dimensions, this function takes three inputs of the data, the *margin* we are interested in applying, and of course, the function to be implemented. We perform this on the entire `mtcars` data set first by columns and then by rows to get the standard deviation. We also first show what the `mtcars` data looks like in its entirety. Notice that it perhaps makes more sense to perform this by columns. Because we are neither car enthusiasts nor particularly mechanically inclined, this last set of code and output may not make much sense:

```
head(mtcars)
                   mpg cyl disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6  160 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6  160 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4  108  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6  258 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8  360 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6  225 105 2.76 3.460 20.22  1  0    3    1
apply(mtcars, MARGIN = 2, FUN = sd)
        mpg         cyl        disp          hp        drat          wt        qsec
vs          am        gear        carb
  6.0269481   1.7859216 123.9386938  68.5628685   0.5346787   0.9784574   1.7869432
0.5040161   0.4989909   0.7378041   1.6152000

apply(mtcars, MARGIN = 1, FUN = sd)
          Mazda RX4       Mazda RX4 Wag          Datsun 710      Hornet 4 Drive   Hornet
Sportabout             Valiant
           53.53888            53.51210            38.86999            79.40933
113.70330            69.95729
         Duster 360           Merc 240D            Merc 230            Merc 280
Merc 280C          Merc 450SE
          122.86626            44.43599            46.68811            57.31739
57.33609            92.42901
         Merc 450SL         Merc 450SLC  Cadillac Fleetwood Lincoln Continental   Chrysler
Imperial            Fiat 128
          92.40957            92.46342           147.34689           145.04382
141.16366            28.06702
        Honda Civic      Toyota Corolla       Toyota Corona    Dodge Challenger         AMC
Javelin          Camaro Z28
          25.10753            26.43470            42.34211            99.84844
96.05271           120.45178
   Pontiac Firebird           Fiat X1-9       Porsche 914-2        Lotus Europa      Ford
Pantera L        Ferrari Dino
          124.56688            27.95293            41.26033            40.34473
123.53038            62.68824
      Maserati Bora          Volvo 142E
          126.32051            44.49888
```

When we want to apply the same function to more than one set of data, we manage a multivariate scenario. In this case, `mapply()` is our function call. Unlike the prior examples, this one takes the function we want to apply as our first variable, and from there takes inputs of strictly positive length. We show the plotted results of the following code in Figure 3-4. As you may recall from *Beginning R*, the `par()` function provides a way to fit multiple graphs into a single image:

```
par(mfrow = c(2, 1))
mapply(plot, mtcars[, c("mpg", "hp")], mtcars[, c("disp", "qsec")])
$mpg
NULL

$hp
NULL
```

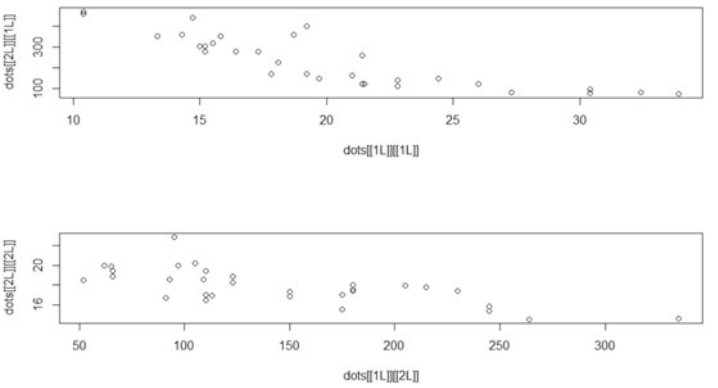


Figure 3-4. *The function plot applied to two subsets of mtcars*

Recursion is a powerful programming idea, and we explore it further in later chapters. For now, consider this example: Suppose a line of people wraps around the corner of a building; you are not sure how long the line stretches. You could step out of your place in line and walk slowly to the front, counting the number of people. This is *regular iteration*—which just like the loops, the `apply` functions have been mostly doing for us. On the other hand, you could keep your place in line and ask the person in front of you, "How many people are in front of you?" If that person repeats your question, eventually it would reach the front of the line, and that person might turn around and say, "Zero!" Moving back through the line to you, it would be possible to get an answer.

Let's Take a look at list `zL`. What if we try `lapply()` on it? This would not be good, because `lapply()` passes each element to `mean()`, and `mean()` cannot handle lists! We show this in the following code, which shows both our list and our first effort to find the arithmetic average:

```
zL <- list( a = list(1, 2, 3:5), b = list(8:12))
zL
$a
$a[[1]]
[1] 1

$a[[2]]
[1] 2

$a[[3]]
[1] 3 4 5

$b
$b[[1]]
[1]  8  9 10 11 12

lapply(zL, FUN = mean)
$a
[1] NA

$b
[1] NA
```

```
Warning messages:
1: In mean.default(X[[i]], ...) :
  argument is not numeric or logical: returning NA
2: In mean.default(X[[i]], ...) :
  argument is not numeric or logical: returning NA
```

The recursive apply function, `rapply()`, is the solution. Depending on the particular results wanted, this function has some interesting options. A check of the help files is highly suggested, although we do not delve deeper into this function at this time.

```
rapply(zL, f = mean)
a1 a2 a3  b
 1  2  4 10
```

It is possible to use "fancier" functions than just `mean()` or `sd()`. In the next chapter, we delve more deeply into what these functions might look like and what they might do. For now, let's take a moment to discuss environment.

Environment is the "world" in which our functions and variables live. We have been mostly residing in the global environment. It is true that our loops have created mini-environments after a fashion. This is why, if we want to keep information available to us after a loop is done, we create those null placeholder variables. We can take a look at our environment and list the variables that are currently in play by using the `environment()` and `ls()` functions. Notice that we are living in the global environment, and most the variables we have been using are present:

```
environment()
<environment: R_GlobalEnv>

ls()
 [1] "forTime"  "forTime2" "i"        "vTime"    "x"        "xCube"    "xL"       "y"
"yL"       "z"        "zL"
```

But of course, this is simply one more list. At times, it may be convenient to manipulate and use this as well. Thus, our last function is `eapply()`. Here we take a look at the class of all objects in our environment; these types should be familiar from prior chapters:

```
eapply(env = environment(), FUN = class)
$xCube
[1] "numeric"

$x
[1] "numeric"

$forTime2
[1] "proc_time"

$y
[1] "numeric"

$z
[1] "numeric"
```

```
$forTime
[1] "proc_time"

$yL
[1] "list"

$vTime
[1] "proc_time"

$i
[1] "numeric"

$xL
[1] "integer"

$zL
[1] "list"
```

Final Thoughts

You have seen several powerful tools for automation—from loops of various sorts that control what happens repeatedly (and indeed, how often), to more specialized tools that are streamlined to cope with the usual suspects in data sets that we are likely to encounter. These techniques are helpful and are often used. However, they are not enough. What if new techniques are developed? How does `mean()` work? What if we need to create our own functions? In the next chapter, we explore functions both in terms of their theoretical framework and their practical applications. The combination of custom functions and loops is the bread and butter of coercing computers to do the busy work for us.

CHAPTER 4

Writing Functions

Writing your own functions in R enables you to combine a set of R commands into a function that is easy to call and can be generalized. Functions are foundational to R. To become a more advanced user or developer of R, a good understanding of what functions are and how to write them is crucial. Broadly speaking, a *function* takes one or more inputs and processes them to produce and return output.

Not every programming task should be converted to a function. However, whenever you find yourself copying and pasting a particular line of your code for the third time, you should likely write a function. Another "no question about it" time to write a function is when your code is over 100 lines or so. Long chunks of code can become almost impossible to read through and understand. Instead, we write functions with good, descriptive names that make our code more readable. Then, on a separate pass, we write code using the functions. The beauty of such a system is that it becomes easier to write focused code inside each function that solves one particular part of your challenge. Another benefit of this approach is that, should greater efficiency ever be required, it is possible to determine which functions are costing you the most processing power or take the longest time to complete. Then research can be done on how to be more efficient in just that spot.

In this chapter, we use the `Hmisc` R package (Harrell Jr, 2016). The following code loads the `checkpoint` (Microsoft Corporation, 2016) package to control the exact version of R packages used and then loads the `Hmisc` package:

```
## load checkpoint and required packages
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(Hmisc)
options(width = 70) # only 70 characters per line
```

Components of a Function

With some exceptions, most functions in R have three components:

- *Formals*: The arguments, or inputs, to the function
- *Body*: The commands that process the input
- *Environment*: The location or context of a function, which determines where it looks for variables

Each can be examined using dedicated functions. First, we write an example function:

```
## create new function, f()
f <- function(x, y = 5) {
  x + y
}
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_4

The function accepts two arguments: x and y. The first argument, x, has no default, but the second argument, y, defaults to a value of 5. Although we can see these easily (in part because we've written the function), the formals, or arguments, of the function can be examined using the `formals()` and `args()` functions:

```
formals(f)
$x

$y
[1] 5

args(f)
function (x, y = 5)
NULL
```

To see the actual R code or the commands used to process the formals, we use the `body()` function:

```
body(f)
{
    x + y
}
```

The commands are always enclosed in opening and closing brackets, `{ }`. In this case, the R code is simply `x + y`, but some functions have hundreds of lines. The last key part of a function is its environment. The environment of a function determines where it looks for variables or objects, which can include both data as well as functions. To see the environment of a function, we use the `environment()` function:

```
environment(f)
<environment: R_GlobalEnv>
```

In this case, the function is in the global environment, where we created the function. For other functions, this would vary. For example, if we look at the environment for the `install.packages()` function, it is the namespace for the `utils` package:

```
environment(install.packages)
<environment: namespace:utils>
```

This provides a common language for discussing R functions. In the remainder of the chapter, we delve deeper into writing functions, and the special code and tools available for use with functions.

Scoping

In R, *scoping* is what determines where to look for a particular variable. Consider, for example, when we type `plot`, how does R translate that code? Where does R look it up? Scope is R's answer for the language idea of context. If I say, "I bought a new bat," you would likely suppose I mean a cricket bat rather than a flying mammal. For R, context comes from the scope and environment. Thus, different environments can lead to different results.

Most aspects of writing R functions are no different than using R interactively. However, one difference is that functions have their own environment. Further, functions often are located in various environments. For instance, when using R interactively, almost all commands are executed from the global environment. In contrast, many functions are written as part of R packages, in which case the function's environment is defined by the R package. Therefore, before jumping in to writing functions, it is helpful to understand scoping. Consider these two examples:

```
plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x00000000190621d0>
<environment: namespace:graphics>

plot <- 5

plot
[1] 5
```

In the first instance, R finds `plot` in the `graphics` package. In the second instance, `R` finds `plot` in the global environment. Note that assigning `5` to the variable, `plot`, does not overwrite the `plot()` function. Instead, it creates another R object with the same name as the function. After creating the new variable, when we type `plot` at the console, R returns the numeric value rather than the function because the assignment, `plot <- 5`, occurs in the *global environment*, which R searches before it checks the environments of different packages. The `search()` function returns the environments in the order that R searches them. Your environment may be different, although there are likely some similarities:

```
search()
 [1] ".GlobalEnv"            "package:Hmisc"
 [3] "package:ggplot2"       "package:Formula"
 [5] "package:survival"      "package:lattice"
 [7] "package:devEMF"        "package:checkpoint"
 [9] "ESSR"                  "package:stats"
[11] "package:graphics"      "package:grDevices"
[13] "package:utils"         "package:datasets"
[15] "package:RevoUtilsMath" "package:methods"
[17] "Autoloads"             "package:base"
```

From the output, we see that R first looks in the global environment (`.GlobalEnv`), then in the package `Hmisc`, and so on until it reaches the `base` package. We can see the current environment by again using the `environment()` function. R always begins looking in the current or local environment. From there, it progresses to the parent environment. We can find the parent environment for a given environment by using the `parent.env()` function:

```
environment()
<environment: R_GlobalEnv>

parent.env(.GlobalEnv)
<environment: package:Hmisc>
attr(,"name")
[1] "package:Hmisc"
attr(,"path")
[1] "C:/Users/Authors/.checkpoint/2016-09-04/lib/x86_64-w64-mingw32/3.3.1/Hmisc"
```

Coming back to functions, each function has its local environment in addition to the function itself being in an environment. The following code shows the local function environment and its parent environment. We can roughly classify variables in functions into one of three types:

- Formal variables
- Local variables defined within the function
- Free or other variables that are neither formals nor local variables

Each is demonstrated in turn with the following code:

```
a <- "free variable"
f <- function(x = "formal variable") {
  y <- "local variable"

  e <- environment()
  print(e)
  print(parent.env(e))

  print(a)
  print(x)
  print(y)
}

f()
<environment: 0x0000000017c846b0>
<environment: R_GlobalEnv>
[1] "free variable"
[1] "formal variable"
[1] "local variable"
```

The variable x is a formal defined as its default value, y is a local variable defined in the body of the function, and a is a variable defined in the function's parent environment. Although it is possible to rely on objects in the search path for a function rather than identified in the formals or as a local variable, it is not a wise idea. Coding to depend on a function's parent environment or the search path can lead to particularly tricky bugs and unexpected behavior. This creates chaos for users or yourself later, when something in the environment seemingly unrelated to the function is changed, and even though it appears the function's code and inputs have not changed, suddenly the output is different.

Although the examples so far have been relatively straightforward, scoping becomes trickier when using nested function calls. In the following code, it may be harder to predict the results of each piece of evaluated code. The next examples set up two functions and then use the functions with three different variables in the global environment:

```
f1 <- function(y = "f1 var") {
  x <- y
  a1 <- f2(x)
  rm(x)
  a2 <- f2(x)
}

f2 <- function(x) {
  if (nchar(x) < 10) {
    x <- "f2 local var"
  }
  print(x)
  return(x)
}

x <- "global var"
f1()
```

```
[1] "f2 local var"
[1] "global var"

x <- "g var"
f1()
[1] "f2 local var"
[1] "f2 local var"

rm(x)
f1()
[1] "f2 local var"
Error in nchar(x) (from #2) : object 'x' not found
```

It is worth experimenting with scoping until it makes sense, because it is necessary for ensuring that the correct object is found. This can be a particular challenge when writing and developing a package or when using functions that appear in multiple packages.

Functions for Functions

In addition to the usual R code and functions that you may use within a function you write, some special functions exist. These are only, or primarily, used within other functions. Even if you are an experienced R user, these may be unfamiliar if you have not previously written functions.

The `match.arg()` function is useful for performing fuzzy matching of arguments. This also has the benefit that if an argument does not match one of the valid options (that is, is invalid), it throws an error. The following code shows two functions that are similar, except one uses the `match.arg()` function. The examples demonstrate fuzzy matching and what happens when an invalid argument is used:

```
f1 <- function(type = c("first", "second")) {
  type
}
f2 <- function(type = c("first", "second")) {
  type <- match.arg(type)
  type
}

f1("fi")
[1] "fi"
f2("fi")
[1] "first"

f1("test")
[1] "test"
f2("test")
Error in match.arg(type) : 'arg' should be one of "first", "second"
```

Argument matching is also useful for ensuring that when a text string is passed, only a valid option is used, and that if it is not, an informative error is thrown. The following code expands on the function we built without `match.arg()` to calculate a mean if `type = "first"`, and a standard deviation if `type = "second"`. However, when an invalid option is passed to the type argument, we get the relatively cryptic error message that `x` is not found:

```
f1b <- function(type = c("first", "second")) {
  if (type == "first") {
    x <- mean(1:5)
  } else if (type == "second") {
    x <- sd(1:5)
  }
  return(x)
}

f1b("test")
Error in f1b("test"): object 'x' not found
```

Another function specific to function arguments is `missing()`. It returns a logical value indicating whether a specific argument is missing from the function call. This is helpful when writing functions that may be used in different ways. The following is an example of a function that calculates Cohen's *d* effect size, which for a single group is defined as the mean of a variable divided by the standard deviation. Cohen's *d* can also be calculated for repeated measures, such as a group measured before and after an intervention. This is done by calculating the difference of the two variables, and then proceeding as before. We use the `missing()` function to determine whether the user passes a single variable, `x`, or two variables, `x` and `y`, so that our function can elegantly handle calculating Cohen's *d* for both one sample and repeated-measures data:

```
cohend <- function(x, y) {
  if (!missing(y)) {
    x <- y - x
  }

  mean(x) / sd(x)
}

cohend(x = c(0.61, 0.99, 1.47, 1.52, 0.45,
             3.34, 1.05, -1.47, 1.3, 0.33),
       y = c(-0.69, 1.6, 0.44, 1, 0.88,
             1.17, 2.4, 1.21, 0.87, 2.15))
[1] 0.09522249
cohend(x = c(0.61, 0.99, 1.47, 1.52, 0.45,
             3.34, 1.05, -1.47, 1.3, 0.33))
[1] 0.796495
```

Also related to determining characteristics of the function call is the function `match.call()`. Whereas `missing()` determines whether a specific argument is missing, and `match.arg()` determines whether an argument matches one of the valid options, `match.call()` captures the entire function call. This might be easier to demonstrate than to explain. The following little function calculates the coefficient of variation, the sample standard deviation divided by the sample mean. It also captures and returns the function call by using `match.call()`:

```
cv <- function(x, na.rm = FALSE) {
  fcall <- match.call()

  est <- sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)

  return(list(CV = est, Call = fcall))
}
```

```
cv(1:5)
$CV
[1] 0.5270463

$Call
cv(x = 1:5)

cv(1:8, na.rm = TRUE)
$CV
[1] 0.5443311

$Call
cv(x = 1:8, na.rm = TRUE)
```

In the output from those two examples, `match.call()` captures exactly the call to the function, though it adds explicit argument names. This can sometimes be useful for keeping a record of exactly what the call was that created particular output. Perhaps the most common place where this is used is in the output from regression models in R. For example, the following code shows a linear model in which the output echoes the function call, which is done using `match.call()`:

```
lm(mpg ~ hp, data = mtcars)

Call:
lm(formula = mpg ~ hp, data = mtcars)

Coefficients:
(Intercept)           hp
   30.09886     -0.06823
```

The `return()` function is typically used at the end of functions to return a specific object, as you have already seen in some of the previous examples, though it was not explicitly discussed. However, `return()` can also be used to return values from any point within a function, thus ending execution of the function. The following function has an `if` statement that, if true, results in early termination of the function. Notice that the final result, if not true, is not even wrapped in `return()`. R returns the last object in a function by default, so an explicit call to `return()` is not strictly necessary:

```
f <- function(x) {
  if (x < 4) return("I'm done!")

  paste(x, "- Fin!")
}

f(10)
[1] "10 - Fin!"
f(3)
[1] "I'm done!"
```

Even though you can use `return()` earlier in a function, this is discouraged, because it can be surprising to users and anyone else reading or debugging code. The same effect as in the preceding code can be accomplished by using flow control:

```
f <- function(x) {
  if (x < 4) {
    "I'm done!"
  } else {
    paste(x, "- Fin!")
  }
}

f(10)
[1] "10 - Fin!"
f(3)
[1] "I'm done!"
```

In addition to not using `return()` midway in functions, some argue that an explicit call to `return()` should not be used at the end of functions either, as it is unnecessary. This remains a point of preference, as it can help draw attention to exactly what is returned at the end of a function.

Although not exclusively used in functions, the `invisible()` function is often used with the object returned by a function. Earlier we made a function that calculates the coefficient of variation and returns the function call. We can modify the function to print the coefficient of variation and invisibly return the rest, as shown in the following code. The use of `invisible()` means that even though the function returns the same object it did before, that object is not shown. The `invisible()` function is perhaps most often used in functions that are designed to create attractive output, such as calls to `summary()` or plotting functions. The function's primary purpose is to show a summary or graph, but in case anyone wants or needs to edit the object, the actual object is invisibly returned and thus can be captured and saved for later use:

```
cv <- function(x, na.rm = FALSE) {
  fcall <- match.call()

  est <- sd(x, na.rm = na.rm) / mean(x, na.rm = na.rm)

  print(est)
  return(invisible(list(CV = est, Call = fcall)))
}

cv(1:8, na.rm = TRUE)
[1] 0.5443311

res <- cv(1:8, na.rm = TRUE)
[1] 0.5443311
res$Call
cv(x = 1:8, na.rm = TRUE)
```

The `on.exit()` function can be used to guarantee that a certain set of commands is executed when the function exits or completes. Expressions in `on.exit()` do execute, even if the function has an error or does not properly complete as expected. An example is shown in the second use case of the little function in the following code. An error causes the function to terminate, and once that happens, the expression in `on.exit()` is executed. Using `on.exit()` is particularly valuable when a function modifies any values outside itself. For example, sometimes a plotting function modifies the default plot parameters and returns them to whatever their original state was on completion. Using `on.exit()` ensures that even if something goes wrong and the function fails or has an error, the user still has all of the original settings:

```
f <- function(x) {
  on.exit(print("Game over"))
  x + 5
}

f(3)
[1] "Game over"
[1] 8

f("a")
Error in x + 5 (from #3) : non-numeric argument to binary operator
[1] "Game over"
```

The last set of function-specific functions we cover is related to giving the software or user a signal. In order of severity, they are `stop()`, `warning()`, and `message()`. The next example is slightly more realistic and calculates the mean of a variable, either on its original scale or on a transformed scale, and then back-transforms the mean. The log scale can be relatively more resistant to outliers. If a log transformation is used, the example code checks whether the variable has any negative values, which are undefined and result in a full error, by using the `stop()` function. The argument passed to `stop()` is the error message to display. A similar process is followed for `warning()`, again with the message to be displayed in the warning. Warnings are different from errors. An *error*, via `stop()`, causes the function to stop being evaluated and terminate. A *warning* is issued at the end, but the function is allowed to continue its evaluation. Finally, `message()` can be used to send a message or signal to the user, but without indicating a real or likely problem, as a warning does. Although not demonstrated, a related convenience function is `stopifnot()`, which allows a logical expression to be passed and issues an error if the expression does not evaluate to a true value. Although this option has the benefit of saving some code, a disadvantage is that you cannot write a custom error message to indicate exactly what went wrong.

One of the reasons it is helpful to use warnings or messages, rather than just calling `print()` or `cat()` to have the function print a message, is that warnings and messages can be (optionally) suppressed by using the aptly named functions `suppressWarnings()` and `suppressMessages()`. All of these functions are demonstrated in the code immediately following:

```
f <- function(x, trans = c("identity", "log")) {
  trans <- match.arg(trans)

  if (trans == "log") {
    if (any(x < 0)) stop("Log is not defined for negative values")
    if (any(x < 1e-16)) warning("Some x values close or equal to zero, results may be
unstable")

    x <- log(x)
    message("x successfully log transformed")

    exp(mean(x))
  } else {
    mean(x)
  }
}

f(c(1, 2, 100))
[1] 34.33333
```

```
f(c(1, 2, 100), trans = "log")
x successfully log transformed
[1] 5.848035

suppressMessages(f(c(1, 2, 100), trans = "log"))
[1] 5.848035

f(c(0, 1, 2, 100), trans = "log")
x successfully log transformed
[1] 0
Warning message:
In f(c(0, 1, 2, 100), trans = "log") :
  Some x values close or equal to zero, results may be unstable

suppressWarnings(f(c(0, 1, 2, 100), trans = "log"))
x successfully log transformed
[1] 0

f(c(-1, 1, 2, 100), trans = "log")
Error in f(c(-1, 1, 2, 100), trans = "log") (from chapter04.R!79246pw#5) :
  Log is not defined for negative valuesFunctionssuppressWarnings()
FunctionssuppressMessages()
```

Debugging

The functions demonstrated so far have all worked or have had purposeful errors that are obvious to spot. Sometimes the process of finding the error, or debugging the code and functions written, takes longer. Fortunately, there are some tools to help the process and some practices to narrow the issues.

Although debugging can apply to any code, not just functions, functions can be particularly tricky to debug without additional tools, because normally all of their code executes without interruption or any chance to see what is happening along the way. In this section, we write a function that uses a formula to calculate the means of a variable by levels of another variable, using `tapply()`, which you previously examined in Chapter 3; the function then plots the raw data with dots for the means, using the following code, as shown in Figure 4-1:

```
meanPlot <- function(formula, d) {
  v <- all.vars(formula)
  m <- tapply(d[, v[1]], d[, v[2]],
              FUN = mean, na.rm = TRUE)

  plot(formula, data = d, type = "p")
  points(x = unique(d[, v[2]]), y = m,
         col = "blue", pch = 16, cex = 2)
}

meanPlot(mpg ~ cyl, d = mtcars)
```

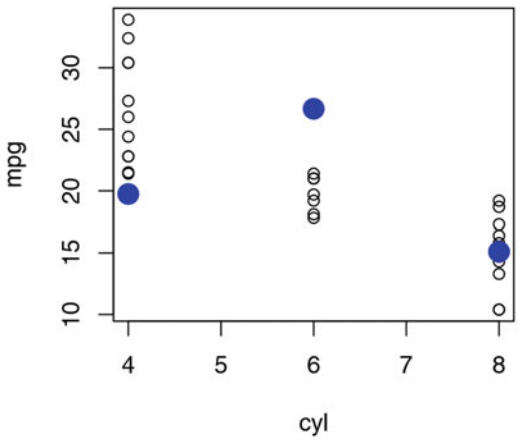


Figure 4-1. *Scatter plot with data in unfilled points, and means in large blue points, showing the incorrect results*

Something does not look correct about the means. The mean for the points when `cyl = 8` looks okay, but the other two seem to fall at extremes. The `debug()` function allows us to debug a function and step through the lines as they are executed. To use it, we first call `debug()` on the function we want to debug:

```
debug(meanPlot)
```

Then when we use the original function, it triggers R to enter debugging. At the first step, R tells us the function call we are debugging in, and where:

```
meanPlot(mpg ~ cyl, d = mtcars)
debugging in: meanPlot(mpg ~ cyl, d = mtcars)
debug at c:/Temp/chapter04.R!79246wk#1: {
    v <- all.vars(formula)
    m <- tapply(d[, v[1]], d[, v[2]], mean, na.rm = TRUE)
    plot(formula, data = d, type = "p")
    points(x = unique(d[, v[2]]), y = m, col = "blue", pch = 16,
        cex = 2)
}
Browse[2]>
debug at #2: v <- all.vars(formula)
Browse[2]>
debug at #3: m <- tapply(d[, v[1]], d[, v[2]], mean, na.rm = TRUE)
Browse[2]>
debug at #5: plot(formula, data = d, type = "p")
Browse[2]> m
       4        6        8
26.66364 19.74286 15.10000
Browse[2]> unique(d[, v[2]])
[1] 6 4 8
Browse[2]> Q
```

Now that we know where the problem occurs, we can fix the function by first sorting the data, so that `tapply()` and `unique()` give the results in the same order. The code is shown here, and the result is in Figure 4-2:

```
meanPlot <- function(formula, d) {
  v <- all.vars(formula)
  d <- d[order(d[, v[2]]), ] ## sorting first
  m <- tapply(d[, v[1]], d[, v[2]],
              FUN = mean, na.rm = TRUE)

  plot(formula, data = d, type = "p")
  points(x = unique(d[, v[2]]), y = m,
         col = "blue", pch = 16, cex = 2)
}

meanPlot(mpg ~ cyl, d = mtcars)
```

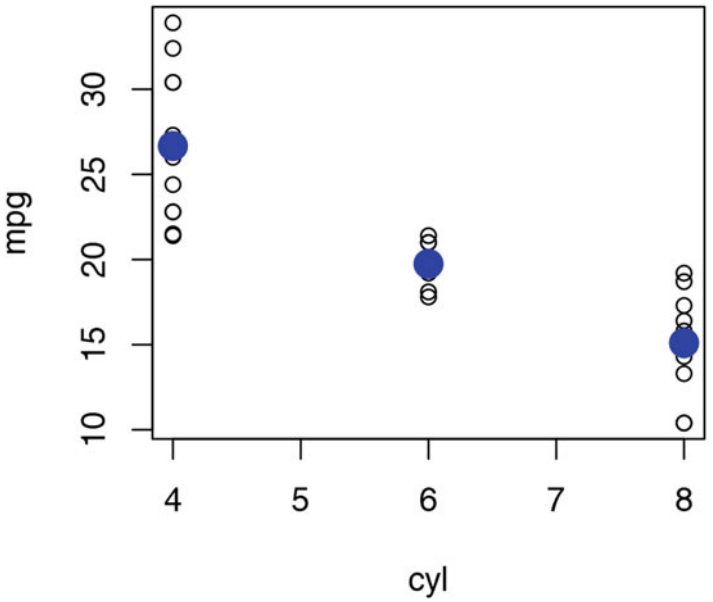


Figure 4-2. *Scatter plot with data in unfilled points, and means in large blue points, showing the corrected results*

If you are debugging your function and do not want to step through each line of code, the `browser()` function can be inserted into the function code. Then, when the function reaches that point, a browser is invoked, and you can examine the current state of variables in the function's local environment. In the following example code, we add a call to `browser()` in the function,examine the current objects available by using `ls()`, examine the contents of the object, `v`, and again type a capital `Q` to quit debugging:

```
meanPlot <- function(formula, d) {
  v <- all.vars(formula)
  d <- d[order(d[, v[2]]), ] ## sorting first
```

```
  m <- tapply(d[, v[1]], d[, v[2]],
              FUN = mean, na.rm = TRUE)

  browser()

  plot(formula, data = d, type = "p")
  points(x = unique(d[, v[2]]), y = m,
         col = "blue", pch = 16, cex = 2)
}

meanPlot(mpg ~ cyl, d = mtcars)
Called from: meanPlot(mpg ~ cyl, d = mtcars)
Browse[1]>
debug at #9: plot(formula, data = d, type = "p")
Browse[2]> ls()
[1] "d"       "formula" "m"       "v"
Browse[2]> v
[1] "mpg" "cyl"
Browse[2]> Q
```

A useful function for debugging when working interactively is `traceback()`. For example, if you call a function such as `lm()` and then get an error message, it can sometimes be difficult to know exactly where or why that mistake occurred. The error is often not even directly from the function you called, as the functions you frequently use may in turn call many other functions internally. The following example shows how this is done by using `traceback()` immediately after the code that resulted in error. The output shows the call stack tracing the path from the initial call to the final code that generated the error. This can be useful information if you need to look at the code to determine why the error occurred:

```
lm(mpg ~ jack, data = mtcars)
Error in eval(expr, envir, enclos) : object 'jack' not found

traceback()
7: eval(expr, envir, enclos)
6: eval(predvars, data, env)
5: model.frame.default(formula = mpg ~ jack, data = mtcars, drop.unused.levels = TRUE)
4: stats::model.frame(formula = mpg ~ jack, data = mtcars, drop.unused.levels = TRUE)
3: eval(expr, envir, enclos)
2: eval(mf, parent.frame())
1: lm(mpg ~ jack, data = mtcars)
```

Finally, sometimes code bugs are not in your code, but in other code you are using, such as from another R package. Although it is rare to find bugs in recommended R packages, it is more frequent in the thousands of other R packages. Also, sometimes problems are not a bug per se, but a difference in how you want to use a function vs. how the original writer envisioned its use. Although the source code for all R packages on CRAN is publicly available for download and editing, it can be a hassle to download an entire package's source code, edit, and reinstall, just to see if doing something slightly different in one function fixes the problem.

Consider the following challenge. Suppose you are using data that sometimes includes infinity for some reason, but you want to include only finite cases. The following code shows an example using the `wtd.quantile()` function from the `Hmisc` package:

```
wtd.quantile(c(1, 2, 3, Inf, NA),
          weights = c(.6, .9, .4, .2, .6))
  0%  25%  50%  75% 100%
 NaN  Inf  Inf  Inf  Inf
```

If we look at the code for `wtd.quantile()`, by typing it into the R console without parentheses, we can see that another function does the main calculations, `wtd.table()`, as shown here:

```
wtd.quantile
function (x, weights = NULL, probs = c(0, 0.25, 0.5, 0.75, 1),
    type = c("quantile", "(i-1)/(n-1)", "i/(n+1)", "i/n"), normwt = FALSE,
    na.rm = TRUE)
{
    if (!length(weights))
        return(quantile(x, probs = probs, na.rm = na.rm))
    type <- match.arg(type)
    if (any(probs < 0 | probs > 1))
        stop("Probabilities must be between 0 and 1 inclusive")
    nams <- paste(format(round(probs * 100, if (length(probs) >
        1) 2 - log10(diff(range(probs))) else 2)), "%", sep = "")
    if (type == "quantile") {
        w <- wtd.table(x, weights, na.rm = na.rm, normwt = normwt,
            type = "list")
        x <- w$x
        wts <- w$sum.of.weights
        n <- sum(wts)
        order <- 1 + (n - 1) * probs
        low <- pmax(floor(order), 1)
        high <- pmin(low + 1, n)
        order <- order%%1
        allq <- approx(cumsum(wts), x, xout = c(low, high), method = "constant",
            f = 1, rule = 2)$y
        k <- length(probs)
        quantiles <- (1 - order) * allq[1:k] + order * allq[-(1:k)]
        names(quantiles) <- nams
        return(quantiles)
    }
    w <- wtd.Ecdf(x, weights, na.rm = na.rm, type = type, normwt = normwt)
    structure(approx(w$ecdf, w$x, xout = probs, rule = 2)$y,
        names = nams)
}
<environment: namespace:Hmisc>
```

Using the same approach, we can examine the `wtd.table()` function. When doing so, we see that although it can automatically remove missing values, it has no check or way to remove nonfinite values. It may seem easier just to change your data, but sometimes data is generated automatically and passed on, so that it is simpler to change a function than it is to modify the data. We can readily copy and paste the code for `wtd.table()` into our R editor and revise it, as shown here:

```
revised.wtd.table <- function (x, weights = NULL, type = c("list", "table"), normwt = FALSE,
    na.rm = TRUE)
{
    type <- match.arg(type)
```

```
    if (!length(weights))
        weights <- rep(1, length(x))
    isdate <- testDateTime(x)
    ax <- attributes(x)
    ax$names <- NULL
    if (is.character(x))
        x <- as.factor(x)
    lev <- levels(x)
    x <- unclass(x)
    if (na.rm) {
        s <- !is.na(x + weights) & is.finite(x + weights)
        x <- x[s, drop = FALSE]
        weights <- weights[s]
    }
    n <- length(x)
    if (normwt)
        weights <- weights * length(x)/sum(weights)
    i <- order(x)
    x <- x[i]
    weights <- weights[i]
    if (anyDuplicated(x)) {
        weights <- tapply(weights, x, sum)
        if (length(lev)) {
            levused <- lev[sort(unique(x))]
            if ((length(weights) > length(levused)) && any(is.na(weights)))
                weights <- weights[!is.na(weights)]
            if (length(weights) != length(levused))
                stop("program logic error")
            names(weights) <- levused
        }
        if (!length(names(weights)))
            stop("program logic error")
        if (type == "table")
            return(weights)
        x <- all.is.numeric(names(weights), "vector")
        if (isdate)
            attributes(x) <- c(attributes(x), ax)
        names(weights) <- NULL
        return(list(x = x, sum.of.weights = weights))
    }
    xx <- x
    if (isdate)
        attributes(xx) <- c(attributes(xx), ax)
    if (type == "list")
        list(x = if (length(lev)) lev[x] else xx, sum.of.weights = weights)
    else {
        names(weights) <- if (length(lev))
            lev[x]
        else xx
        weights
    }
}
```

Unlike the original `wtd.table()` function, the revised function works exactly as we want:

```
wtd.table(c(1, 2, 3, Inf, NA),
          weights = c(.6, .9, .4, .2, .6))
$x
[1]   1   2   3 Inf

$sum.of.weights
[1] 0.6 0.9 0.4 0.2

revised.wtd.table(c(1, 2, 3, Inf, NA),
          weights = c(.6, .9, .4, .2, .6))
$x
[1] 1 2 3

$sum.of.weights
[1] 0.6 0.9 0.4
```

However, the challenge is that the `wtd.quantile()` function and other `Hmisc` functions that use `wtd.table()` still do not work as we hope, because they do not use our revised function. Assigning our function to the name `wtd.table()` in the global environment is not sufficient, because scoping rules mean that `Hmisc` functions access the `wtd.table()` function first from the `Hmisc` environment, not from our global environment. It is like having a file of the same name on your computer, but in two different folders. For all the `Hmisc` functions that utilize `wtd.table()` to use our revised function, we need not only to name it correctly, but also to put it in the correct place. We can assign an object to a specific namespace by using the `assignInNamespace()` function:

```
assignInNamespace(x = "wtd.table",
                  value = revised.wtd.table,
                  ns = "Hmisc")

wtd.quantile(c(1, 2, 3, Inf, NA),
          weights = c(.6, .9, .4, .2, .6))
   0%   25%   50%   75%  100%
2.000 2.225 2.450 2.675 2.900

wtd.Ecdf(c(1, 2, 3, Inf, NA),
          weights = c(.6, .9, .4, .2, .6))
$x
[1] 1 1 2 3

$ecdf
[1] 0.0000000 0.3157895 0.7894737 1.0000000
```

Although the `assignInNamespace()` function has no output, we can see that afterward the `wtd.quantile()` function and others that depend on `wtd.table()`, such as `wtd.Ecdf()`, are now working as we hoped. A caveat about `assignInNamespace()` is that you cannot assign any object to an object that does not already exist in that namespace (that is, you can overwrite only existing objects). You also are not allowed to use `assignInNamespace()` in any packages you may want to submit to CRAN. Even if it is sometimes the convenient approach, it would become confusing if people did this as a general rule. Were this to happen, the definition of functions in package A would depend on whether you had loaded package B, because package B could overwrite (not just mask) the function definition in package A. Note that any changes you make in this way are temporary and vanish when you restart R. However, this can be a helpful technique for debugging an existing package, as it is a relatively easy way to make sure that the issue encountered using `wtd.quantile()` was indeed "fixed" by the suggested change to `wtd.table()`. At this point, if it were truly a bug or even if it was just a desirable feature, you could e-mail the package maintainer to suggest the change, confident that the suggested code works. To see the current maintainer, you can just type `maintainer("Hmisc")`, or whatever the package of interest is called.

Summary

We covered a lot of functions and a lot about functions in this chapter! In case you need to refresh your memory about any specific functions, Table 4-1 lists the key functions introduced in this chapter and provides a brief description of each. Of course, you can look up more in the official help files.

Although the formal names of various parts of a function are not necessarily critical, learning how to use and write functions efficiently may be one of the best investments in learning R you ever make. Writing functions provides a way out of writing repetitive code. There are no strict rules, but if you find yourself doing the same task often, there is a good chance it is worth writing a function to do that. It might be a big task involving many pieces, or it might be a small task. For example, in psychology, it is common to report values at "high" and "low" values of a continuous variable, which are often defined as mean +/– 1 standard deviation. This is easy to do in R by using the `mean()` and `sd()` functions. If you do it a lot, it may be worth writing a short, one-line function so that rather than type `mean(x) + sd(x)`, you just type `msd(x)`, or whatever you call your function. If you do not feel comfortable playing with functions and writing your own, it would be a good idea to get some more practice before moving on to Chapters 5 and 6, where we assume you are comfortable with functions.

Table 4-1. *Key Functions Described in This Chapter*

Function	What It Does
`formals()`	Allows you to see the formal arguments of a function you build.
`args()`	Shows default values and names for a created function's arguments.
`body()`	Shows the body of a function (the code between { and }).
`environment()`	Shows the environment your function lives in (often the global environment, so far).
`search()`	Shows the search order for functions. Remember, people may have already used your function name!
`parent.env()`	Takes an environment and tracks it up one level.
`match.arg()`	Allows for fuzzy matching of function arguments.
`missing()`	Tests whether a value was passed to the function. Note that this will be false after `match.arg()`.
`match.call()`	Captures the entire function call; used often in regression.
`return()`	Not required, possibly contentious, and for sure, if used be the last part of a new function.
`invisible()`	Suppresses output.
`on.exit()`	Regardless of a successful or failed function attempt, this executes its argument.
`stop(), stopifnot(), warning(), message()`	These are errors of various levels of severity, from full-on stop and an error, to a milder warning, to a mostly polite message.
`suppressWarnings(), suppressMessages()`	These two do precisely what they say. Once you're familiar with a function, warnings or messages may be tedious or safely ignored.
`debug()`	Use this to start debugging a function that is not working properly.
`browser()`	This goes into a function body, right before the part you want to debug.
`traceback()`	Provides a list of expressions leading to the source of an error.
`assignInNamespace()`	Allows for the temporary replacement of a function with a locally crafted function.
`maintainer()`	Called on a package, shows a contact name and e-mail to go to for troubleshooting.

CHAPTER 5

Writing Classes and Methods

It is often helpful to have a function behave differently depending on the type of object passed. For example, when summarizing a variable, it makes sense to create a different summary for numeric or string data. It is possible to have a different function for every type of object, but then users would have to remember many function names, and to remain unique, function names may be longer. Object-oriented programming (OOP) is based on objects and is implemented in R (as in most programming languages) by using two concepts: classes and methods. A *class* defines a template, or blueprint, describing the variables and features of an object as well as determining what methods work for it. For example, a house may be defined as having a floor, four walls, a roof, and a door. Specific data represents these properties, such as the dimensions and color of each wall. The *methods* are behaviors or actions that can be performed on a particular object type. For instance, a house can be painted, which changes its color, but a house cannot be eaten. R has three object-oriented systems: S3, S4, and R5. This chapter covers the S3 and S4 systems, which are the most common.

In this chapter, we use the `ggplot2` R package (Wickham, 2009). The following code loads the `checkpoint` (Microsoft Corporation, 2016) package to control the exact version of R packages used and then loads the `ggplot2` package:

```
## load checkpoint and required packages
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(ggplot2)
options(width = 70) # only 70 characters per line
```

S3 System

The *S3 system* is the most common object-oriented system. It is also the easiest to start using and the simplest of the systems. The S3 system is easy to use in part because it is quite informal, and mostly focused on the functions or methods. These advantages are also limitations, as the S3 system provides no formal framework for ensuring that objects meet the requirements for a class. For a more in-depth guide to R programming using the S3 system, see *S Programming* by W.N. Venables and B.D. Ripley (Springer, 2004).

S3 Classes

In R, some types, or *classes*, of objects are available by default, and almost all can be thought of as vectors or generic vectors. For example, matrices and arrays are essentially vectors with attributes indicating the dimensions. Lists are generic vectors, in which each element of the vector may contain another vector. Data frames are lists in which each item is a vector, but all with equal length, and thus have a tabular format.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_5

Vectors can hold specific types of data, such as logical, integer, numeric (real numbers), or character strings. These fundamental objects and classes are provided by the base package. S3 classes are created by building on the objects and classes provided by the base package.

■ **Note** S3 classes are created by using regular R objects (for example, vectors or lists) and classes (for example, logical or numeric). S3 classes are defined by setting the class name via `class(object) <- "class name"`. Elements of S3 objects are typically accessed by using `` `$` ``, `` `[` ``, or `` `[[` ``.

The practical creation of S3 classes is simple. First, create an R object that meets the requirements or characteristics of the class, and then define the S3 classes by labeling the object with the class name. Because S3 classes differ from other R objects only in having special names or attributes, they are accessed and manipulated the same way as other basic R objects, by using `` `$` ``, `` `[` ``, and `` `[[` ``. To check the class of an object, we can use the `class()` function. In the following example, the `mtcars` object is queried to determine that it has a data frame class:

```
class(mtcars)
[1] "data.frame"
```

To see how the class of an object is set, we can look at many of the functions from base R, such as `table()`. Here, we print the source code for the `table()` function, leaving some of the middle off to save space:

```
table
function (..., exclude = if (useNA == "no") c(NA, NaN), useNA = c("no",
    "ifany", "always"), dnn = list.names(...), deparse.level = 1)
{
[ommitted for space]
    y <- array(tabulate(bin, pd), dims, dimnames = dn)
    class(y) <- "table"
    y
}
<bytecode: 0x0000000017fa8180>
<environment: namespace:base>
```

The object returned at the end, `y`, is an array, but has a `table` class. The object class is set using the idiom `class(object) <- "class name"`. It is also possible to assign more than one class to an object. This is done in the much the same fashion as assigning a single class and in order of preference. The following example stores the results from calling `table()` in the object `x`, and then sets two classes, first `newclass` and then the original `table` class:

```
x <- table(mtcars$cyl)
class(x) <- c("newclass", "table")
class(x)
[1] "newclass" "table"
```

The value of assigning multiple classes is a sort of backup, primarily for methods. For example, if you create a new class that is a variant of a table or data frame, you may write a dedicated method for printing, but rely on methods the original class for other functions, such as plotting or summaries.

The types of classes you can write are virtually endless. A simple way to start is by creating special or augmented cases of existing classes. In the following example, we make a special case of a data frame, with x and y coordinates and text labels, called `textplot`. The simplest way to "create" the class is to create a data frame and then change its class. If we use only our new class label, calling the `print()` function results in the default method, but if we label it with both our new `textplot` label and as a data frame secondly, then calling `print()` falls back to the method for data frames:

```
d <- data.frame(
  x = c(1, 3, 5),
  y = c(1, 2, 4),
  labels = c("First", "Second", "Third")
)
class(d) <- "textplot"

print(d)
$x
[1] 1 3 5

$y
[1] 1 2 4

$labels
[1] First  Second Third
Levels: First Second Third

attr(,"row.names")
[1] 1 2 3
attr(,"class")
[1] "textplot"

class(d) <- c("textplot", "data.frame")

print(d)
  x y labels
1 1 1  First
2 3 2 Second
3 5 4  Third
```

In the S3 system, there is no formal way to create an object from a particular class. However, the most common way to create objects of a specific class is as the output from a function. Dedicated functions can be written to create an object of a specific class, or functions can be written that perform operations and output results as an object of a specific class. The latter approach is more typical in R when using the S3 system.

When you are creating a new class, some of the desired features or elements may be present in an existing class. If this is the case, it may make sense to build on, or extend, an existing class. For instance, data frames build on lists, requiring that each element of the list be a vector with the same length. Likewise, the `textplot` class we created previously builds on data frames, requiring three elements named `x`, `y`, and `labels`. In this instance, we would say that `textplot` *inherits* from `data.frame`. That is, `textplot` is a child of the parent class, `data.frame` (note that parent classes are also referred to as the *super class*, or *base class*). If a class inherits from only one other class, it is called *single inheritance* (that is, it has only one parent class). If a class inherits from multiple classes, it is called *multiple inheritance* (that is, it has more than one parent class). If a class inherits from another class, it may inherit features of the data stored, and it may inherit methods, the functions that operate on objects of a specific class. Inheriting methods are especially useful to avoid re-creating the wheel.

■ **Note** *Inheritance* refers to creating a new class by building on the features and methods of an existing class. The new class is known as the *child class*, and the classes from which the new class is derived are the *parent*, or *super*, classes. A child class may inherit both features of the parent class and use of the parent class's methods.

In the next example, we write a little function to use the formula interface to build a `textplot` object from a data frame. The function has two arguments: the formula, called `f`, and the data, called `d`. The first part of the code uses the `stopifnot()` function introduced in Chapter 4 to check that the classes of the objects passed to the function match what is expected. To test the classes, we use the `inherits()` function, which assesses whether a particular object is, or inherits from, the specified class. For example, our `textplot` class is secondly a data frame, and so testing that uses `inherits(object, "data.frame")` would evaluate to `TRUE`. When we build S4 classes, we see a more formal definition and system for class inheritance. The next part of the function gets all the variables from the formula, in order, from the specified data frame, renames the columns, applies our `textplot` class, and returns the object:

```
textplot_data <- functions(f, d) {
  stopifnot(inherits(d, "data.frame"))
  stopifnot(inherits(f, "formula"))

  newdata <- get_all_vars(formula = f, data = d)
  colnames(newdata) <- c("y", "x", "labels")
  class(newdata) <- c("textplot", "data.frame")

  return(newdata)
}

## example use
textplot_data(f = mpg ~ hp | cyl, d = mtcars[1:10, ])
                     y   x labels
Mazda RX4         21.0 110      6
Mazda RX4 Wag     21.0 110      6
Datsun 710        22.8  93      4
Hornet 4 Drive    21.4 110      6
Hornet Sportabout 18.7 175      8
Valiant           18.1 105      6
Duster 360        14.3 245      8
Merc 240D         24.4  62      4
Merc 230          22.8  95      4
Merc 280          19.2 123      6
```

These examples show how easy it is to use the S3 system to create classes. Next, we explore how to write methods for existing or new classes.

S3 Methods

Methods are functions or operations that can be performed on objects of specific classes. Even if you do not write your own classes, you may write your own methods. Writing S3 methods is like writing functions as we did in Chapter 4. The only difference is that S3 methods have a special naming convention and require a generic function for users, which takes care of dispatching to the appropriate method.

■ **Note** S3 methods are regular R functions that follow a specific naming convention: `foo.classname()`. Users call the generic function, `foo()`, which dispatches to the appropriate method based on the class of object passed in as an argument. If no generic function exists, a generic must be written that includes a call to `UseMethods()`, which handles the actual method dispatch.

To start, let's write a simple plotting method for the `textplot` object class we developed. To make an S3 plot method, we use the function name—here, `plot`—followed by the class name—`function.classname()` or, in this case, `plot.textplot()`. As long as we name our function in that way, it works as an S3 method, as long as a generic `plot()` function exists, a topic we discuss shortly.

Our plot function is shown in the following code. The call to the `par()` function adjusts the default margins for a graph, to reduce excess white space on the top and right of our graph. The results are stored in the object, `op`, as this stores the original graphical parameters. Then the user's original graphical parameter state can be restored when the function exits by calling `par(op)` on exit, a function you learned about in Chapter 4. Next, we create a new plot area by calling `plot.new()`, and set the dimensions of our new plot by calling `plot.window()` with the x and y limits determined by the range of the data. Next, we plot the labels by using the `text()` function, which takes the coordinates and labels. Finally, we add an axis on the bottom, `side = 1`, and left, `side = 2`. The original `textplot` data object is returned invisibly at the end.

```
plot.textplot <- function(d) {
  op <- par(mar = c(4, 4, 1, 1))
  on.exit(par(op))

  plot.new()
  plot.window(xlim = range(d$x, na.rm = TRUE),
              ylim = range(d$y, na.rm = TRUE))
  text(d$x, d$y, labels = d$labels)

  axis(side = 1, range(d$x, na.rm = TRUE))
  axis(side = 2, range(d$y, na.rm = TRUE))

  invisible(d)
}
```

Next, we need to make some data and then `plot()` it. Note that because it is a method, we do not need to call our function by its full name, `plot.textplot()`. We can simply call `plot()`, and R takes care of dispatching the data object to the correct method based on the object class, `textplot`. The result is shown in Figure 5-1, and the code is shown here:

```
dat <- textplot_data(f = mpg ~ hp | cyl, d = mtcars[1:10, ])
plot(dat)
```

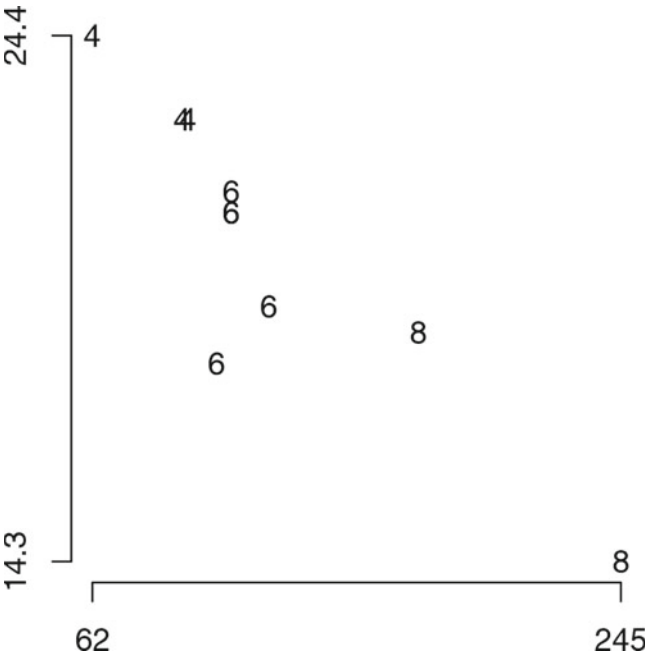


Figure 5-1. *The plot of text labels at specific coordinates, demonstrating the use of custom methods for custom classes*

This example shows just how easy it is to use the S3 system. Almost no special functions or effort is required. Just create a regular R object however you want, add a custom class label, write a function, and give it a special name—and that is essentially all that is required. However, there are a few special functions and tools for S3 methods. To start, let us see what happens when we look at the source code for `plot()`. This generic function contains just three arguments. The body consists only of a call to `UseMethod()`, which is what tells R to check the argument classes and dispatch to an appropriate method. If you are writing methods and a generic function does not exist, you also need to write the generic function. Writing a generic function in the S3 system is a straightforward task that requires only considering the default arguments to include:

```
plot
function (x, y, ...)
UseMethod("plot")
<bytecode: 0x0000000017ee3bd0>
<environment: namespace:graphics>
```

To see the methods available, we use the `methods()` function. The result shows many specific methods available for `plot()`, including our newly written `plot.textplot`:

```
methods(plot)
 [1] plot,ANY-method               plot,color-method
[omitted for space]
 [71] plot.table*                   plot.textplot
[omitted for space]
 [81] plot.varclus                  plot.xyVector*
see '?methods' for accessing help and source code
```

Some of the functions are followed by an asterisk, indicating that these methods are not public or cannot be directly accessed, such as `plot.table()`. For example, if we type its name, we get an error that it cannot be found:

```
plot.table
Error: object 'plot.table' not found
```

■ **Note** The `::` operator is used to refer to publicly exported functions from a specific package, primarily when multiple packages export functions with the same name: `package::foo()`. The `:::` operator is used to access functions from a package's namespace that are not exported. Use nonpublic functions with caution, as they are subject to change without notice.

Functions that are not public or have not been exported from a package namespace can still be accessed as methods. These function also can be accessed directly by specifying the package they are from and using the `:::` operator, revealing the function source code:

```
graphics:::plot.table
function (x, type = "h", ylim = c(0, max(x)), lwd = 2, xlab = NULL,
    ylab = NULL, frame.plot = is.num, ...)
{
    xnam <- deparse(substitute(x))
    rnk <- length(dim(x))
    if (rnk == 0L)
        stop("invalid table 'x'")
    if (rnk == 1L) {
        dn <- dimnames(x)
        nx <- dn[[1L]]
        if (is.null(xlab))
            xlab <- names(dn)
        if (is.null(xlab))
            xlab <- ""
        if (is.null(ylab))
            ylab <- xnam
        is.num <- suppressWarnings(!any(is.na(xx <- as.numeric(nx))))
        x0 <- if (is.num)
            xx
        else seq_along(x)
        plot(x0, unclass(x), type = type, ylim = ylim, xlab = xlab,
            ylab = ylab, frame.plot = frame.plot, lwd = lwd,
            ..., xaxt = "n")
        localaxis <- function(..., col, bg, pch, cex, lty) axis(...)
        if (!identical(list(...)$axes, FALSE))
            localaxis(1, at = x0, labels = nx, ...)
    }
    else {
        if (length(dots <- list(...)) && !is.null(dots$main))
            mosaicplot(x, xlab = xlab, ylab = ylab, ...)
```

```
        else mosaicplot(x, xlab = xlab, ylab = ylab, main = xnam,
            ...)
    }
}
<bytecode: 0x00000000053765058>
<environment: namespace:graphics>
```

In addition to writing methods for new classes, it is sometimes helpful to write methods for existing classes. For example, the popular `ggplot2` package has no default method for working with a linear model or regression objects. To start, we set up a simple regression model by using the built-in `mtcars` data. From there, we are predicting `mpg` from `hp`, `vs`, their interaction (all of which are created from `hp * vs`, which expands to the two main effects and their interaction or product term), and `cyl` dummy coded, through the call to `factor()`. The results are shown here by calling `summary()` on the object:

```
m <- lm(mpg ~ hp * vs + factor(cyl), data = mtcars)
summary(m)
Call:
lm(formula = mpg ~ hp * vs + factor(cyl), data = mtcars)

Residuals:
    Min      1Q  Median      3Q     Max
-4.7640 -1.4424 -0.1703  1.5882  6.9382

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  26.92908    2.71758   9.909 2.56e-10 ***
hp           -0.01519    0.01554  -0.978  0.33718
vs            8.53352    4.95297   1.723  0.09678 .
factor(cyl)6 -4.21121    1.93887  -2.172  0.03916 *
factor(cyl)8 -8.65096    2.69738  -3.207  0.00354 **
hp:vs        -0.09101    0.04363  -2.086  0.04692 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 3.007 on 26 degrees of freedom
Multiple R-squared:  0.7913,  Adjusted R-squared:  0.7511
F-statistic: 19.71 on 5 and 26 DF,  p-value: 4.181e-08
```

We can check the class of the object and that there is no current `ggplot()` method by using the following code:

```
class(m)
[1] "lm"
methods(ggplot)
[1] ggplot.data.frame* ggplot.default*    ggplot.summaryP
[4] ggplot.transcan
see '?methods' for accessing help and source code
```

Although the `fortify()` function has a method for linear models that creates a data frame suitable for use with `ggplot()`, it extracts only the raw data, fitted values, and residuals. To show the effects of a model, it can be helpful to plot predicted values as a function of one variable holding other variables at specific values. The method that follows implements a system to do this.

It takes the same basic arguments as `ggplot()` but adds the `vars` argument, and each variable used in the model is passed with a specific set of values to hold it at for prediction. All possible combinations of these are created by passing the list as arguments to the `expand.grid()` function by using the `do.call()` function. The dependent variable, or `yvar`, is then extracted from the model formula and converted to a character. New predictions along with standard errors for the predictions are generated. A 95 percent confidence interval is generated with the lower limit, LL, and upper limit, UL, based on the fit or predicted value and the normal quantiles times the standard error of the fit. Finally, the column name is changed from `fit` to whatever the dependent variable's actual name was, and then the data for prediction and the predicted values are combined into a new data frame. This is passed to `ggplot()`, which dispatches to the `ggplot()` method for data frames.

```
ggplot.lm <- function(data, mapping, vars, ...) {
  newdat <- do.call(expand.grid, vars)
  yvar <- as.character(formula(data)[[2]])
  d <- as.data.frame(predict(data, newdata = newdat, se.fit = TRUE))
  d <- within(d, {
    LL <- fit + qnorm(.025) * se.fit
    UL <- fit + qnorm(.975) * se.fit
  })
  colnames(d)[1] <- yvar
  data <- cbind(newdat, d[, c(yvar, "LL", "UL")])
  ggplot(data = data, mapping = mapping, ...)
}
```

With this method in place, we can easily make some graphs from our linear regression model. The following code uses our new method, specifying the exact values to hold predictor variables, adds a line by using `geom_line()`, and uses a black-and-white theme with `theme_bw()`. The result is shown in Figure 5-2; the code is as follows:

```
ggplot(m, aes(hp, mpg), vars = list(
                          hp = min(mtcars$hp):max(mtcars$hp),
                          vs = mean(mtcars$vs),
                          cyl = 8)) +
  geom_line(size=2) +
  theme_bw()
```

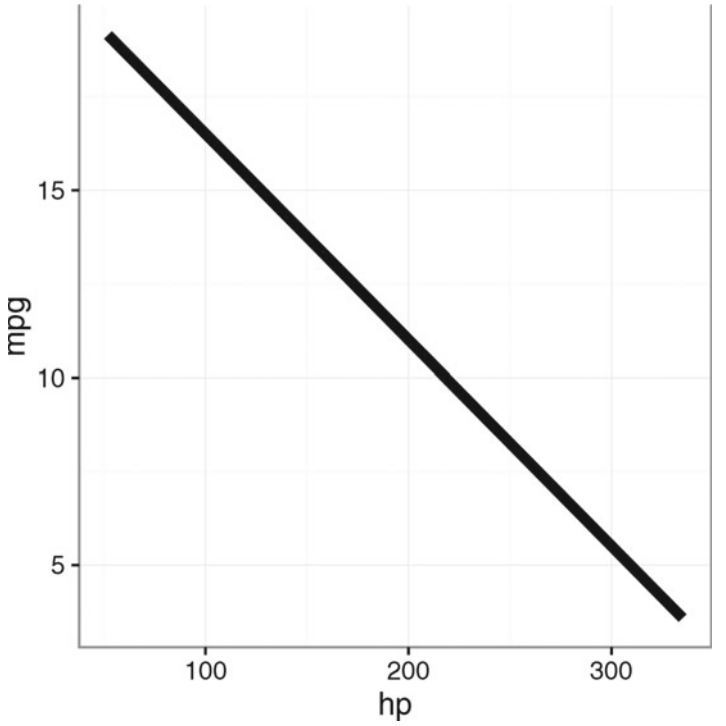


Figure 5-2. *Predicted regression line from the model, using the ggplot.lm() method*

Because all possible combinations of the predictor values are created by using `expand.grid()` in our method, we can make several predicted lines, such as holding `vs` at `0` and `1`, shown in the following example code and in Figure 5-3:

```
ggplot(m, aes(hp, mpg, linetype = factor(vs), group = factor(vs)), vars = list(
                        hp = min(mtcars$hp):max(mtcars$hp),
                        vs = c(0, 1),
                        cyl = 8)) +
  geom_ribbon(aes(ymin = LL, ymax = UL), alpha = .25) +
  geom_line(size=2) +
  theme_bw()
```

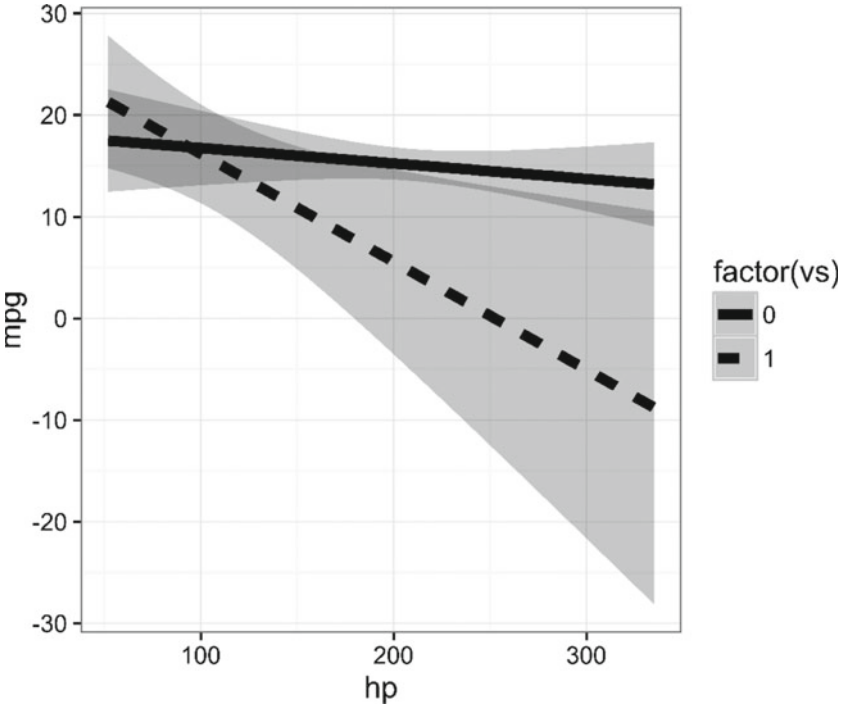


Figure 5-3. *Predicted regression lines from the model, with confidence intervals, using the ggplot.lm() method*

These examples provide a simple introduction to what can be done when writing new classes and methods or extending existing classes and methods. Because the S3 system is the most commonly used in R, it may be the single most important system to learn; it can be used so widely and can extend many classes of objects.

S4 System

In contrast with the S3 system, the *S4 system* is a formal system. The benefit of this formality is a greater assurance that objects of a particular class contain exactly what is expected. The downside of the formality is a greater complexity: more functions required to set up classes and methods, and more-rigid requirements in the programming. Whereas the S3 system allows for writing quick-and-dirty code, the S4 system requires more careful planning and comes with higher overhead.

Objects have three components: classes, slots, and methods. The name and structure of the object—what it contains—is the *class*. The variables or other objects stored in the object are *slots*. Finally, the functions that can operate on an object are its *methods*. Throughout this section, we go over each of these components. For further reading on the S4 system, one excellent guide is *Software for Data Analysis: Programming with R* by John Chambers (Springer, 2010).

S4 Classes

Because S4 classes are more formal, planning is required before writing a new class. Unlike S3 classes, in S4 the names and types of every variable to be included must be specified. Previously we defined a class, `textplot`, by using the S3 system. We can define the same class by using the S4 system. The `x` and `y` arguments should both be numeric type, and the `labels` argument should be a character string. We also know that the length of all the arguments should be the same. Finally, we may want to assert that although it is okay to have a missing label, a blank label is not allowed. All of this needs to be considered and specified in advance—because although in the S3 system none of this could be controlled, in the S4 system we can define everything when we create the class. To create our first S4 class, we use the `setClass()` function, shown in the next example. Not every argument is required, but it is considered a good practice to be as explicit as possible when defining a new class.

■ **Note** S4 classes are created by calling `setClass()`. The `Class` and `slots` arguments are required. The class name is specified as a character string, and slots (holding variables) are defined as a named character vector; names correspond to slot names, and values correspond to the class of each slot. An "empty" object can be specified by using the `prototype` argument, and validity checking can be set by passing a function to the `validity` argument.

The first argument, `Class`, provides the name of the class. Next, the `slots` are defined by using a named vector in which the names indicate the slot names and the values indicate the type of each slot. The `prototype` argument is not required, but it is helpful to define, as it determines how to create an "empty" object, or how an object of that class is created when no specific data is specified. The `validity` argument is also not required, but allows explicit checks to be run that the object conforms to expectations. R by default ensures the appropriate type of objects is passed to the slots, but many other tests and validity checks can be added to reduce the chances of an object being created that does not work as intended. Here is the `setClass()` example:

```
setClass(
  Class = "textplot",
  slots = c(
    x = "numeric",
    y = "numeric",
    labels = "character"),
  prototype = list(
    x = numeric(0),
    y = numeric(0),
    labels = character(0)),
  validity = function(object) {
    stopifnot(
      length(object@x) == length(object@y),
      length(object@x) == length(object@labels))
    if (!all(nchar(object@labels) > 0, na.rm = TRUE)) {
      stop("All labels must be missing or non zero length characters")
    }
    return(TRUE)
  }
)
```

To create new objects of a particular class in the S4 system, we use the function `new()`. In the code that follows, we examine three attempts to create a new object. First, we make a correct and valid object. Then we look at what happens if we try to use the wrong type of argument. Finally, we look at an example that tests the validity function we created.

```
new("textplot",
    x = c(1, 3, 5),
    y = c(1, 2, 4),
    labels = c("First", "Second", "Third"))
```

```
An object of class "textplot"
Slot "x":
[1] 1 3 5

Slot "y":
[1] 1 2 4

Slot "labels":
[1] "First"  "Second" "Third"
```

```
new("textplot",
    x = c(1, 3, 5),
    y = c(1, 2, 4),
    labels = 1:3)
```

```
Error in validObject(.Object) :
  invalid class "textplot" object: invalid object for slot "labels" in class "textplot": got
class "integer", should be or extend class "character"
```

```
new("textplot",
    x = c(1, 3, 5),
    y = c(1, 2, 4),
    labels = c("First", "Second", ""))
```

```
Error in validityMethod(object) (from #16) :
  All labels must be missing or non zero length characters
```

These errors would not happen if we were using the S3 system to define a class, as no such type checking nor validity checking occur. In the previous examples, each attempt to create a new object had at most one error. The next example has two errors: the vector passed to the y slot is not the same length as the other two, and there is a zero-length character label. However, as currently written, only the first error is caught, because the validity function stops as soon as there are any problems:

```
new("textplot",
    x = c(1, 3, 5),
    y = c(1, 2),
    labels = c("First", "Second", ""))
Error: length(object@x) == length(object@y) is not TRUE
```

Another way to write the validity function is so that, rather than throwing errors for any problems, the function collects them and returns all at the end. While revising the validity function, we could also think about how to make the error messages more informative. As it stands, it is fairly straightforward to see what the problem is, but perhaps not to see *exactly* what the problem is. For example, it is evident that the lengths are not equal, but is it that x is too long or y is too short? This is implemented in the revised code to define a class.

However, before diving into improving the validity function, we need a brief diversion on creating and formatting character strings in R. First, when writing text, new lines can be inserted by including the special character, `\n`. To see the examples, we use the `cat()` function, which stands for *concatenate*, and print, and writes text out to the R console (or other locations if a file is specified). The following two examples are identical except for the line break between the `a` and `b` in the second example:

```
cat("ab", fill = TRUE)
ab

cat("a\nb", fill = TRUE)
a
b
```

One commonly used function is `paste()`, which can combine vectors or collapse them. Combining is shown in the first example that follows. The two vectors are combined by using the separator, defined by the `sep` argument, an empty string in our case. In the second example, a single vector with multiple elements is collapsed into a single character string. How the elements of the vectors are combined into one string is determined by the argument to `collapse`—in our example, the line break character.

```
paste(c("a", "b"), c(1, 2), sep = "")
[1] "a1" "b2"

paste(c("a", "b"), collapse = "\n")
[1] "a\nb"
```

These are useful functions for us when writing a validity-checking function, as they allow us to combine multiple errors into one string with line breaks as needed.

The other key function we use is `sprintf()`. Its first argument is a user-defined string, with special symbols that always start with the percentage sign (`%`) where values should be substituted. The subsequent arguments are the values to substitute. An example may be the clearest way to show it. Here, `%d` is used to indicate that an integer is substituted, and then R substitutes in 98, 80, and 75. The order of substitution is the order of appearance.

```
sprintf("First (%d), Second (%d), Third (%d)", 98, 80, 75)
[1] "First (98), Second (80), Third (75)"
```

Commonly used format options for substitutions are `%d` for integers, `%f` for fixed-point decimals, `%s` for strings, and `%` for a literal percentage sign. Each is demonstrated in this next example, and further documentation is available in the help pages, `?sprintf`. For the numeric value, we use `0.2` to specify that the number should be rounded to two decimal places.

```
sprintf("Integer %d, Numeric %0.2f, String %s, They won by 58%%",t
        5, 3.141593, "some text")
[1] "Integer 5, Numeric 3.14, String some text, They won by 58%"
```

Armed with `paste()` and `sprintf()`, we can proceed to revise the validity function for our `textplot` class to provide more-informative errors, and to run all checks, collecting errors along the way and returning all of them at the end. One final note: previously, we used the idiom `new("classname", arguments)` to create a new object of a particular class. While this is perfectly acceptable, there is a shortcut. The function `setClass()` is primarily called for its side effect of defining a new class, but it also invisibly returns a

constructor function. If we save the results from our call to `setClass()`, by convention in an object with the same name as the class, we can use the resulting object to create a new object of that class:

```
textplot <- setClass(
  Class = "textplot",
  slots = c(
    x = "numeric",
    y = "numeric",
    labels = "character"),
  prototype = list(
    x = numeric(0),
    y = numeric(0),
    labels = character(0)),
  validity = function(object) {
    errors <- character()
    if (length(object@x) != length(object@y)) {
      errors <- c(errors,
                  sprintf("x (length %d) and y (length %d) are not equal",
                          length(object@x), length(object@y)))
    }
    if (length(object@x) != length(object@labels)) {
      errors <- c(errors,
                  sprintf("x (length %d) and labels (length %d) are not equal",
                          length(object@x), length(object@labels)))
    }
    if (!all(nchar(object@labels) > 0, na.rm = TRUE)) {
      errors <- c(errors, sprintf(
        "%d label(s) are zero length. All labels must be missing or non zero length",
        sum(nchar(object@labels) == 0, na.rm = TRUE)))
    }

    if (length(errors)) {
      stop(paste(c("\n", errors), collapse = "\n"))
    } else {
      return(TRUE)
    }
  }
)
```

Now when we create the same object with multiple problems, we get far more information and save some keystrokes. We can see the lengths of `x` and `y`, and also learn that there are problems with the labels:

```
textplot(
  x = c(1, 3, 5),
  y = c(1, 2),
  labels = c("First", "Second", ""))
Error in validityMethod(object) (from #30) :

x (length 3) and y (length 2) are not equal
1 label(s) are zero length. All labels must be missing or non zero length
```

S4 Class Inheritance

So far, you have seen how to define new S4 classes. It may still seem like using the S4 system requires much more work than the S3 system, and with little benefit, aside from more formal validation and error checking. One of the powerful features of the S4 system is inheritance.

■ **Note** S4 classes can inherit from existing S4 classes by calling `setClass()` with the additional argument `contains = "S4 Class to Inherit"`. Existing slots, prototype, and validity checking are inherited. Only new slots and corresponding prototypes/validity checking need to be specified in the new `setClass()` call.

We previously created a simple `textplot` class. Now suppose that although sometimes our simple class is sufficient, at other times we may need more. For example, at times we may want to drill down into the data and create a panel of plots for different subsets of the data by some grouping variable. To this end, we want a `groupedtextplot` class. However, the only additional data we need is one more slot. One option is to copy and paste our old code and then modify it as needed. In this section, we explore how using inheritance lets us reuse and extend existing classes. When a class inherits from another class, all of the slots from the previous class are also inherited, as is validity checking. Next, we create our `groupedtextplot` class. It is similar to creating a new class, but we define only new slots, and then specify the inheritance by using the argument `contains`. Because the validity checking is also inherited, we need to specify only validity checks for the new slots.

```
groupedtextplot <- setClass(
  Class = "groupedtextplot",
  slots = c(
    group = "factor"),
  prototype = list(
    group = factor()),
  contains = "textplot",
  validity = function(object) {
    if (length(object@x) != length(object@group)) {
      stop(sprintf("x (length %d) and group (length %d) are not equal",
                   length(object@x), length(object@group)))
    }
    return(TRUE)
  }
)
```

With that small amount of code, we are ready to use our new class. In the following two examples, the first shows a correctly created new object, and the latter shows the familiar error messages when we attempt to create an invalid object:

```
gdat <- groupedtextplot(
    group = factor(c(1, 1, 1, 1, 2, 2, 2, 2)),
    x = 1:8,
    y = c(1, 3, 4, 2, 6, 8, 7, 10),
    labels = letters[1:8])
gdat
An object of class "groupedtextplot"
Slot "group":
```

```
[1] 1 1 1 1 2 2 2 2
Levels: 1 2

Slot "x":
[1] 1 2 3 4 5 6 7 8

Slot "y":
[1]  1  3  4  2  6  8  7 10

Slot "labels":
[1] "a" "b" "c" "d" "e" "f" "g" "h"
```

```
groupedtextplot(
    group = factor(c(1, 1, 1, 1, 2, 2, 2, 2)),
    x = 1:8,
    y = c(1, 3, 4, 2, 6, 8, 7),
    labels = c(letters[1:7], ""))
Error in validityMethod(as(object, superClass)) (from #30) :

x (length 8) and y (length 7) are not equal
1 label(s) are zero length. All labels must be missing or non zero length
```

In this case, `textplot` would be called the *parent class*, and `groupedtextplot` would be called the *child class*. This relationship can be diagrammed (and sometimes for complex inheritance, diagramming is helpful). By convention, the relationship is shown graphically with an arrow pointing from the child to the parent(s), such as `textplot <- groupedtextplot`. Also by convention, parents are typically on the left, or above if graphing from top to bottom. Although we cover inheritance from only a single parent in this book, classes can inherit from multiple parents, and those parents can inherit from parents, and so on. It is in these cases where a visual diagram is particularly helpful. Another benefit of using inheritance, rather than writing a whole new class, is that methods are also inherited. This means that we can reuse both the slots from the parent as well as the methods written for the parent class, a topic we turn to in the next section.

S4 Methods

In addition to the `methods()` function you saw earlier to display the methods available for a given function, in the S4 system we can find methods by using `showMethods()`. Because of the more formal class system, `showMethods()` can be used to show all methods for a specific function, or to show all methods (for any function) for a particular class by using the `classes = "class name"` argument. This can be helpful if you are working with a new class and want to know what methods have already been written and are available.

■ **Note** Available S4 methods can be examined by using `showMethods("generic function")`. New S4 methods are defined by calling `setMethod()`. The main three arguments are `f`, `signature`, and `definition`, containing the name of the generic function (string), the S4 class name that will dispatch to this method (string), and a function that is the actual method, respectively.

To write new methods in the S3 system, we simply write functions with a special naming convention. To define S4 methods, we use the function `setMethod()`. It is possible to write new methods for existing classes, as well as writing methods for new classes, of course. For a new class, a method is needed for `show()`. When an object is simply typed at the console, R shows it by calling `show()`. Without a `show()` method for a new class, the default printing is quite ugly, as you have seen in our example so far.

In the following code, we define a new method for our `textplot` class. The first argument is the function name for which we create a method. The next argument, the signature, is the name of the class. In this case, because `show()` takes a single argument, only one class name needs to be specified. For functions with multiple arguments, the signature can become more complex, with different methods depending on the class of multiple arguments. Finally, we write our function, the definition of the method. The code is relatively simple, using the `cat()` function to display the values. The argument `fill = TRUE` has the effect of adding a line feed so that each line starts with `X:` or the variable label, and then the values. The `head()` function is used to get at most the first five values, or less if there are fewer values.

```
setMethod(
  f = "show",
  signature = "textplot",
  definition = function(object) {
    cat("     X: ")
    cat(head(object@x, 5), fill = TRUE)
    cat("     Y: ")
    cat(head(object@y, 5), fill = TRUE)
    cat("Labels: ")
    cat(head(object@labels, 5), fill = TRUE)
  })
[1] "show"
```

R echoes the name of the generic function, `show()`, for which we just created a method. Now we get some nicer output when we create a `textplot` class object:

```
dat <- textplot(
  x = 1:4,
  y = c(1, 3, 5, 2),
  labels = letters[1:4])

dat
     X: 1 2 3 4
     Y: 1 3 5 2
Labels: a b c d
```

Once we start defining methods, the benefits of class inheritance become even greater. Because `groupedtextplot` inherits from `textplot`, if no method is defined for `groupedtextplot`, it falls back to the method for `textplot`, if available. Although not perfect, because the grouping is not shown, this is still nicer than the default:

```
gdat
     X: 1 2 3 4 5
     Y: 1 3 4 2 6
Labels: a b c d e
```

Next, we define a method for the `[` function, which is used to subset data. This function is more complex, as we must build a logic tree allowing several possible combinations of arguments. The first argument of the function, `x`, is for the object to be subset. By convention, `i` refers to rows or observations, and `j` to columns or variables. If only `i` is not missing (that is, rows or observations are specified), typically all variables are included. If only `j`(variables) is specified, typically all observations are included. If both are

specified, only select variables and select observations are included. This is accomplished by using a series of `if` and `else if` statements. There are three new functions:

- `validObject()` is a generic function that executes the validity check, if present, for a specific object class.
- `slotNames()` returns a character vector giving the names of each slot.
- `slot()` works similarly to the `@` operator, but can use character strings to extract slots by name.

After the method is set, several examples of its uses are shown:

```
setMethod(
  f = "[",
  signature = "textplot",
  definition = function(x, i, j, drop) {
    if (missing(i) & missing(j)) {
      out <- x
      validObject(out)
    } else if (!missing(i) & missing(j)) {
      out <- textplot(
        x = x@x[i],
        y = x@y[i],
        labels = x@labels[i])
      validObject(out)
    } else if (!missing(j)) {
      if (missing(i)) {
        i <- seq_along(x@x)
      }

      if (is.character(j)) {
        out <- lapply(j, function(n) {
          slot(x, n)[i]
        })
        names(out) <- j
      } else if (is.numeric(j)) {
        n <- slotNames(x)
        out <- lapply(j, function(k) {
          slot(x, n[j])[i]
        })
        names(out) <- n[j]
      } else {
        stop("j is not a valid type")
      }
    }

    return(out)
  })

dat[]
     X: 1 2 3 4
     Y: 1 3 5 2
Labels: a b c d
```

```
dat[i = 1:2]
     X: 1 2
     Y: 1 3
Labels: a b

dat[j = 1]
$x
[1] 1 2 3 4

dat[j = "y"]
$y
[1] 1 3 5 2

dat[i = 1:2, j = c("x", "y")]
$x
[1] 1 2

$y
[1] 1 3
```

With a `show` and `subsetting` method defined for our `textplot` class, we can easily leverage those to make a `show` method for our `groupedtextplot` class by looping through the object by group, subsetting, and showing each subset:

```
setMethod(
  f = "show",
  signature = "groupedtextplot",
  definition = function(object) {
    n <- unique(object@group)
    i <- lapply(n, function(index) {
      cat("Group: ", index, fill = TRUE)
      show(object[which(object@group == index)])
    })
  })

gdat
Group:  1
     X: 1 2 3 4
     Y: 1 3 4 2
Labels: a b c d
Group:  2
     X: 5 6 7 8
     Y: 6 8 7 10
Labels: e f g h
```

Summary

This chapter has introduced the S3 and S4 systems in R for developing new classes and methods. The S4 system, in particular, can be complicated, with inheritance from multiple parent classes and methods that are specialized to the class of more than one argument. Even if you use and develop in R, you may only rarely develop new classes, as there are already classes for most data types. However, it can often be helpful to develop or extend existing methods, and at the very least, understanding how classes and methods work makes it easier to use existing ones.

Hopefully, this chapter is enough to get you started using these systems and to see their capabilities. Table 5-1 describes key functions covered in this chapter. In Chapter 6, we bundle functions and classes and methods together to make our own R package. The focus is on making an R package, so you do not need to be too comfortable with classes and methods. If your work would benefit from the use of the S4 system, you can get slightly more in-depth coverage from, "How S4 Methods Work" by John Chambers (available online at `http://developer.r-project.org/howMethodsWork.pdf`). If you intend to work with the S4 system and need an in-depth dive, we recommend *Software for Data Analysis: Programming with R*. Another good book more focused on general R programming and the older S3 system is *S Programming*.

Table 5-1. *Key Functions Described in This Chapter*

Function	What It Does
`class()`	Returns the class of an object (S3/S4).
`inherits()`	Checks whether an object is of a certain class or inherits from that class.
`methods()`	Returns a list of the available methods for a given function.
`showMethods()`	Returns a list of the available methods for a given function or, if using the `classes` argument, available methods for any function for a given class.
`:::`	Operator that allows access to nonexported (nonpublic) functions from a package.
`function.class()`	Generic scheme for naming an S3 method, using the function name, followed by a period, followed by the class of object it should be applied to.
`setClass()`	Defines a new S4 class.
`setMethod()`	Defines a method for a particular function for a particular S4 class.
`@`	Low-level way to access slots by name in objects in the S4 system, similar to $ for other R objects or S3 class objects.
`new()`	Creates a new S4 object of a specific class.
`paste()`	Pastes strings together. Can operate on several vectors or collapse together all the elements of a single vector.
`sprintf()`	Formats strings and allows for substituting numbers and other strings into a defined template. Useful for making informative error messages or other messages to users.
`show()`	Generic function to show an R object. Also, the default function that is called when you type an object name at the R console.
`` `[`() ``	Operator/generic function used to subset data or to access specific variables or rows. In the S4 system for a new class, methods must be defined, or the function is not usable.
`head()`	Lists the first few elements or rows of an object.
`validObject()`	Generic function that checks when an object with an S4 class is valid by using the validity checks specified when the class was created (if any). Called by default when an object is created, but can also be called explicitly after modifying an object to check whether it is still valid.
`slotNames()`	Returns all the slot names of an S4 class object, similar to `names()` for other R objects or S3 class objects.
`slot()`	Can be used to access a specific slot of an S4 object.

CHAPTER 6

Writing a Package

Packages are the fundamental way to document, share, and distribute R code and data. Our goal is to write our own packages, and our fair warning is that this chapter is particularly complex because of the many software tools that are employed during R package development.

In this chapter, we make sure of four packages. The `devtools` package (Wickham and Chang, 2016) provides functions to help set up, document, and manage the development of an R package. The `roxygen2` package (Wickham, Danenberg, and Eugster, 2015) greatly eases writing documentation for R packages by allowing the function documentation to be written inline next to functions by using comments. We also use two other, less critical packages. The `testthat` package (Wickham, 2011) is not required but has functions that facilitate quality control by testing that functions return expected values. Similarly, the `covr` package (Hester, 2016) facilitates quality control by testing the percentage of package code that is executed (covered) by tests. Ideally, 100 percent of code would be covered by tests, though in practice, many R packages have no tests, so any coverage is better than average. In the package, we use one of the methods from Chapter 5 for `ggplot2`, so we also include the now familiar `ggplot2` (Wickham, 2009) package.

Writing packages can be a complex process. It may be helpful to familiarize yourself with the final product, available online at https://github.com/ElkhartGroup/AdvancedRPkg. We also recommend referring to our official GitHub record of the package as you go through this chapter, if you are unsure whether a file is in the correct location or set up correctly. The following code loads the `checkpoint` (Microsoft Corporation, 2016) package to control the exact version of R packages used and then loads the required package:

```
## load checkpoint and required packages
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(testthat)
library(devtools)
library(roxygen2)
library(covr)
library(ggplot2)
options(width = 70) # only 70 characters per line
```

Before You Get Started

This chapter covers some of the basics of writing an R package, and also introduces tools to facilitate the processes not included in the official manual, *Writing R Extensions*, available online at https://cran.r-project.org/doc/manuals/R-exts.html. The manual can be quite technical, but it is the definitive guide. It is required reading if you plan to submit R packages to CRAN, and a good idea to read even if you just plan to develop packages for your own use or a more limited user base such as your company or lab group.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_6

Many R packages contain code from other programming languages such as C, C++, Fortran, or Java. Often this is because other languages, such as C, can be compiled and optimized to run much faster than R, so package developers may choose to write computationally intensive parts of their code in a compiled and highly efficient language. However, for this chapter, we discuss how to write only R packages containing pure R code or data, as the process is nearly identical, with most differences being idiosyncratic to the language included (for example, makefiles).

Before developing R packages, some tools are required. Indeed, for those not used to software development, writing R packages can require a rather daunting toolchain. If you are using a Linux system, chances are you have many of these already or can readily install them. In Windows and Mac OS, it can take a bit more setup. Regardless of the system, the tools need to be accessible, and this means adding them to the system path so that when called from a command line, they can be found.

If using Windows, the main tools required (some command-line tools as well as compilers) are available from a binary installer at https://cran.r-project.org/bin/windows/Rtools/. Before the final next of the install, there are two check boxes, one of which may be unchecked and adds this to your system path. Just select both. Also, you need LaTeX, a typesetting system. MiKTeX is a popular choice (www.miktex.org/); be sure to select the 64-bit option if your system is 64 bit. While the default options for these two pieces are enough in most situations, we suggest that MiKTeX users change to Yes the option for installing default packages on the fly. Further details on the required toolchain are available from the R manual at https://cran.r-project.org/doc/manuals/R-admin.html#The-Windows-toolset.

If using Mac OS, get Xcode, developer tools, from the App Store. You may also need to get XQuartz (https://xquartz.macosforge.org/). As on Windows, you need LaTeX. MacTeX is one choice (www.tug.org/mactex/). Additional details are available from the R manual (https://cran.r-project.org/doc/manuals/R-admin.html#OS-X).

Two more decisions need to be made to progress to writing our first R package. First, what to name the R package? This seemingly simple task is complicated by the fact that over 8,000 R packages are currently on CRAN, and the number is growing rapidly (https://cloud.r-project.org/web/packages/available_packages_by_name.html). Although you can give a package that is not to be uploaded to CRAN the same name as one on CRAN, doing so is problematic if you ever use the CRAN package or any other package that depends on it. Even if your package is not to be submitted to CRAN, someone else in the future might write a package with the same name and submit it to CRAN. For these reasons, choosing the package name requires some thought. Throughout this chapter, all the examples are to build one R package, called `AdvancedRPkg`. The second decision is what license to use for the package. If you're planning to submit to CRAN, the license must be compatible with CRAN. Even if the package is never sent to CRAN, a license may still be relevant. For this chapter, we use the GPLv3 license.

Version Control

Up until now, we have discussed relatively casual code development, some basic programming, functions, and classes. Developing new R packages is more complex, with many more files to manage. With this greater complexity, it is helpful to have an efficient system for backing up files and for going back to earlier versions. Version control systems provide an excellent way both to back up code and to roll back changes to a prior version (for example, if changes to implement a new feature break an old feature or introduce a bug). Various version control systems exist, but perhaps the most popular now is the open source Git originally developed to provide version control for the Linux kernel. Git can be used directly from the command line or through a graphical interface, of which there are several. Many people use GitHub (https://github.com/), a service that uses Git and hosts repositories online, freely for public projects. GitHub also provides a graphical interface for Windows and Mac OS (https://desktop.github.com/). However, if you do not want to learn or use Git or another version control system, feel free to ignore this section as well as any commands related only to version control throughout this chapter.

We use Git (and GitHub) for Windows in this chapter to provide version control for the example R package we develop. Specifically, this book uses the Windows GitHub desktop client v3.3.0. Git repositories can be used solely on a local computer, but using them with GitHub makes it easy to collaborate on packages, and share early package code with others.

Setting up a Git repository is possible from the command line, but perhaps the most intuitive way for new users to create a new Git repository is online, directly through GitHub. We make a new repository for the R package we develop in this chapter and call it `AdvancedRPkg`. The repository is initialized with a `README` file, which we'll edit shortly. We can tell Git to ignore certain files or files with particular extensions by adding them to a file called `.gitignore`. Each file or extension is listed on its own line. We can edit it later, but for now, we just have GitHub create one. Figure 6-1 shows the steps to do this.

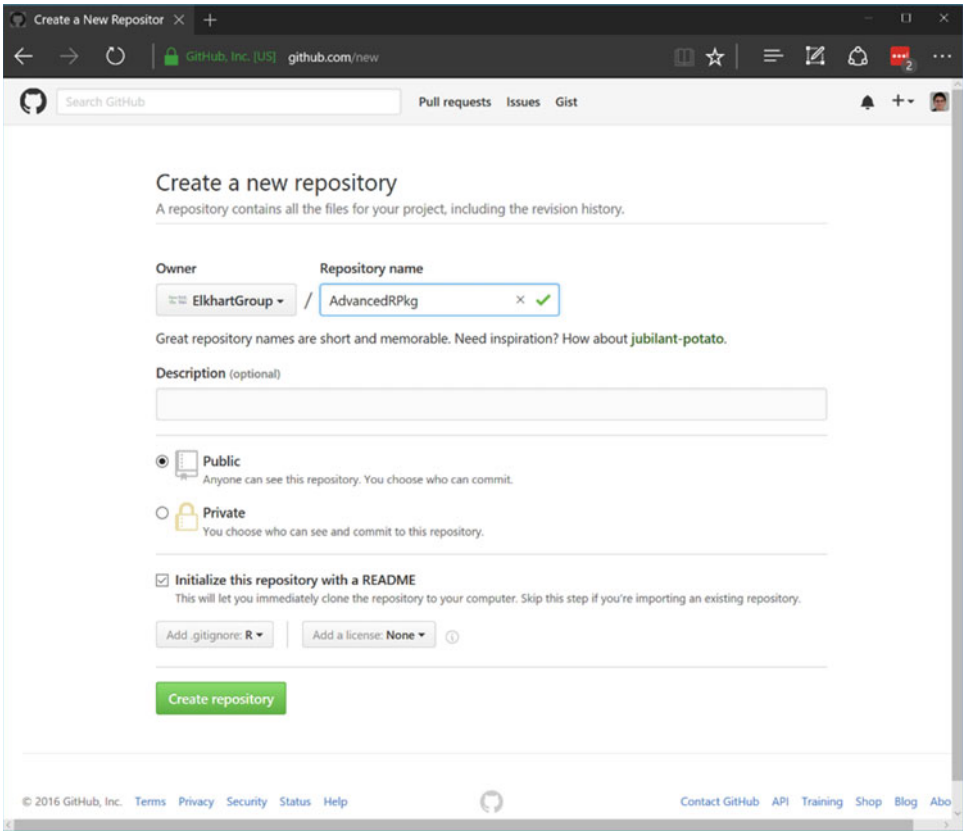


Figure 6-1. *GitHub page for creating the AdvancedRPkg repository*

If all works as planned, the results should look something like Figure 6-2. The repository is empty, except for the `README` and `.gitignore` files.

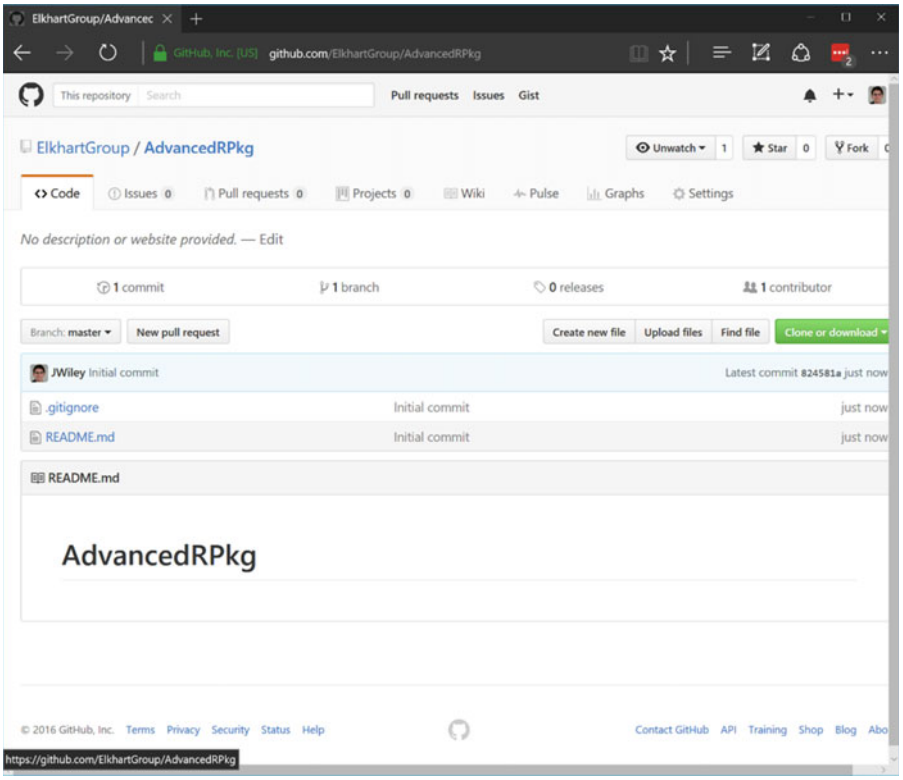


Figure 6-2. *GitHub page showing the created AdvancedRPkg repository*

From here, we can clone the repository to our local computer. Essentially, this creates a local copy, where we do our package writing. To do this, open the GitHub desktop client, click the + sign, select Clone, and then pick the repository, as shown in Figure 6-3.

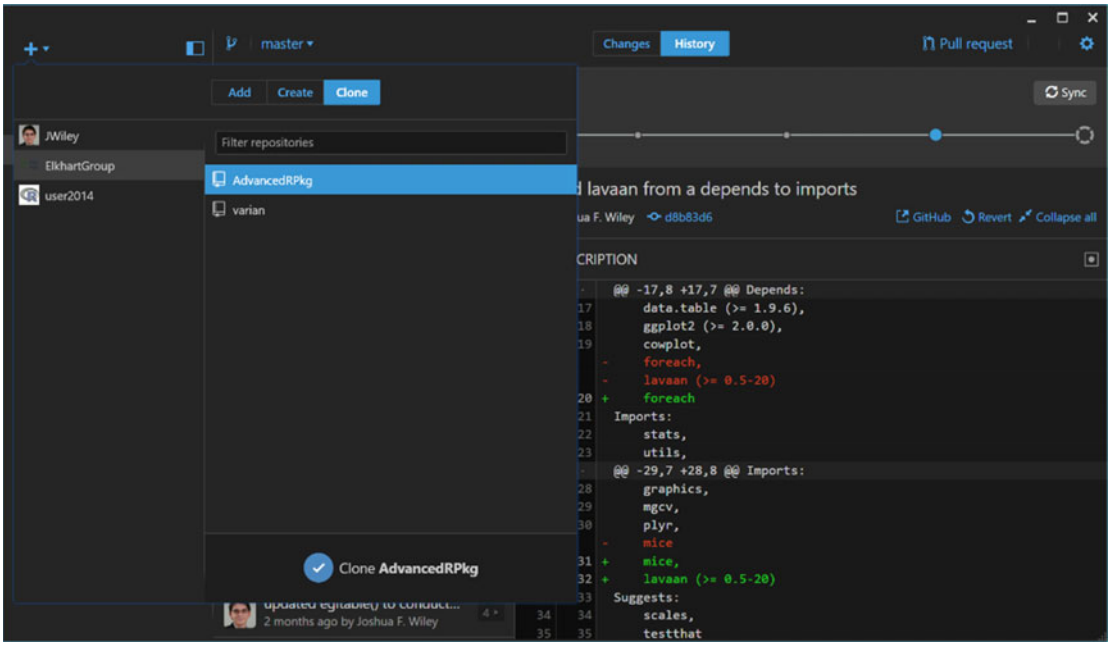


Figure 6-3. *GitHub desktop client cloning the AdvancedRPkg repository*

After clicking Clone, we see an option to select a directory into which the repository should be cloned. Note that whatever directory you choose, a new directory is created for the repository with the same name as the repository (that is, `AdvancedRPkg`). Settings for the repository can be managed via the terminal, or by clicking the gear icon in the GitHub desktop client and choosing Repository Settings. One of them shows the currently ignored files. Files are ignored based on the `.gitignore` file, and because we selected R earlier when creating the repository, some default file extensions that are often desirable to ignore have been included by default (Figure 6-4).

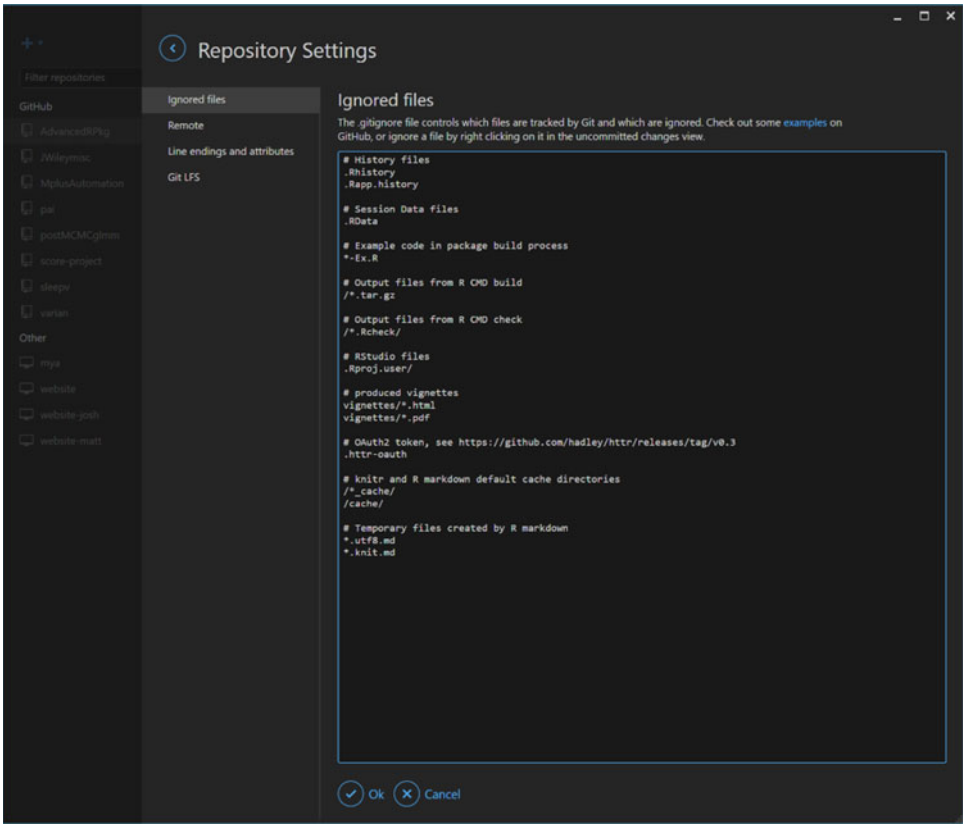


Figure 6-4. *GitHub desktop client showing settings for the AdvancedRPkg repository, including various file extensions that are ignored by the Git repository*

Windows often creates temporary (sometimes hidden) files, `desktop.ini`, in directories. To avoid including these, we can edit the Repository settings by adding a new line at the end and typing `desktop.ini`. Then these files are ignored. *Ignored* files are those that the Git repository does not track and store changes in over time. They continue to exist in the directory; Git simply does not monitor and version them.

Aside from setting up a new Git repository, the tasks you are likely to do with Git are to commit changes to a repository and sync your local copy of the repository with the online (GitHub) version. Commits can be thought of as taking a snapshot of the state of files at the time of the commit. To avoid redundancy commits, snapshot only the changes made to files since the previous commit. Although even a slightly modified binary file may be difficult to compare, plain-text files, like R code, are easy to compare. Git is extremely efficient at tracking changes in text files over time and allowing you to either see what changes were made or go back to specific points in the past. Access to this history of a project (repository) is particularly helpful if errors or bugs are introduced into the code, so it can be pinpointed exactly when the bug was introduced (for example, how many results based on the code may be impacted?) and see previous working versions of code.

In slightly more complex use cases, Git can be effectively used to manage different versions of the code. For example, it is common to have a project and periodically release stable or production-ready versions of the code, while at the same time continuing development. This is accomplished in Git by using different branches of the same repository, and periodically merging some of the changes from one branch (say, the development branch) into the stable branch.

We show a few more relevant Git commands throughout this chapter. For more background on using Git, a great and free resource is *Pro Git* by Scott Chacon and Ben Straub (Apress, 2014), available at https://progit.org/. For questions and answers, Stack Overflow (http://stackoverflow.com) is a good resource. It is likely someone else has already asked a question similar to yours, and if not, you can ask and get answers. Also note that if you use RStudio for your R code, it has some integration with Git built in. For more on this, see RStudio Support at https://support.rstudio.com/hc/en-us/articles/200532077-Version-Control-with-Git-and-SVN. Finally, note that you can access the online repository publicly at https://github.com/ElkhartGroup/AdvancedRPkg.

R Package Basics

An *R package* is just a directory with a particular set of files. Although the core of an R package is, of course, R code, many of the files are not strictly R code. Table 6-1 lists the primary files that may be in the root of an R package directory. Not all of these files are required, depending on other characteristics of the package. We'll get help creating some of these files by using the `devtools` package later in this chapter.

Table 6-1. *Files That May Be Used in the Root Directory of an R Package.*

File	Description
`DESCRIPTION*`	Provides general information about the package. Required fields include Package (package name), Version, License, Title (package description), Author, and Maintainer (may be the same or different from the author). Also often includes information on dependencies, unless the package is stand-alone.
`NAMESPACE*`	Controls the package namespace, including which objects to export and which to import from other packages. Although required, we generate this automatically by using the `roxygen2` package.
`README/README.md`	Not required and ignored by R, but helpful for readers and users. Provides general information on the package (where to get help, a brief overview, installation guidance, or whatever else would be useful in a brief document).
`NEWS`	Provides information on changes or news about the package. Commonly includes new features and bug fixes as well as any other major changes compared to previous versions.
`LICENSE`	A license file if one of the common licenses is not used or if additional information is required.
`INDEX`	Optional, as generated automatically from the documentation files. But if specified, provides a listing of all interesting/useful objects as a name and description on each line.
`configure`, `cleanup`	Optional Bourne shell scripts that are run before and after installation, respectively, on Unix systems (for example, Linux, Unix).

** indicates a required file*

In addition to the root-level files, numerous subdirectories can be included in an R package. These are listed in Table 6-2, along with brief descriptions. For our relatively simple package, we work with only a handful of these, including `R`, `man`, and `tests`.

Table 6-2. *R Package Subdirectories*

Subdirectory	Description
`R*`	Directory that contains all the R source code for functions, classes, methods, and so forth.
`man*`	Directory that contains the R documentation files. We generate these automatically by using the `roxygen2` package.
`tests`	Optional directory containing R code used to test that the package works as expected.
`data`	Optional directory containing data to be shipped with the package. Typically, included data sets are small and used to illustrate a package's features, rather than as a primary means of sharing data.
`demo`	Optional directory containing demonstrations of the R package, as R source code files.
`exec`	Can contain additional required executable scripts. Only files are included, not subdirectories.
`inst`	Can contain additional files that are copied to the installation directory when the package is installed. For example, may be used to share a `NEWS` file with users or to include images or other nonstandard documentation.
`po`	Used to add translations of C and R error messages and other localization-related tasks.
`src`	Typically, source code from other languages, such as C, C++, or Fortran. This code is usually compiled; and if you use it, it often requires specifying a makefile.
`tools`	Not commonly used, but can be used to provide additional files required for configuration.
`vignettes`	Used to provide one or more vignettes or guides to using the package. Vignettes typically have more introductory background and complete examples of how you might use a package overall, compared with function documentation, which is generally unique to that function. Though not required, can be very helpful for users.

** indicates a required subdirectory*

Starting a Package by Using DevTools

At a minimum, at the root directory of your package, you need `DESCRIPTION` and `NAMESPACE` files. You also need a subdirectory, `R`, which, sensibly, is where you put your R code. This is not enough for a package to submit to CRAN, but it is sufficient to start a functional package for private use. However, even for private use, proper documentation is incredibly helpful. The directory and file structure of packages is regrettably complex, especially to describe in text. Thus as a reference, the following is the final directory structure of our package that you should have by the end of this chapter, obtained from the within the `AdvancedRPkg` directory:

```
.
├── data
│   └── sampleData.rda
├── DESCRIPTION
├── man
│   ├── ggplot.lm.Rd
│   ├── meanPlot.Rd
│   ├── sampleData.Rd
│   └── textplot-class.Rd
├── NAMESPACE
├── R
│   ├── plot_functions.R
│   ├── sampledata.R
│   └── textplot.R
├── README.md
└── tests
    ├── testthat
    │   └── test_textplot.R
    └── testthat.R

5 directories, 13 files
```

For the initial setup, we use the function, `setup()`. We can specify more information, but the minimum is the path. We also set `rstudio = FALSE` so that the code works with any editor, not just RStudio. To see what has been created, we can use the `list.files()` function, which shows that `DESCRIPTION` and `NAMESPACE` files and an `R` directory have been created. For the following code, either run it from the R command line or make a new R file (we called ours `chapter06.R`). Note that it is important to adjust the path argument to point to the directory with the Git repository, wherever you cloned that on your local machine. In our case, R's working directory (which you can check by calling the `getwd()` function) is in the directory directly above the `AdvancedRPkg` directory. If your R session is not there, either adjust the path argument or change R's working directory to be in the parent directory to `AdvancedRPkg` by using the `setwd()` function:

```
setup(
  path = "AdvancedRPkg/",
  rstudio = FALSE)
Creating package 'AdvancedRPkg' in '~/Apress_AdvancedR/RFiles'
No DESCRIPTION found. Creating with values:

Package: AdvancedRPkg
Title: What the Package Does (one line, title case)
Version: 0.0.0.9000
Authors@R: person("First", "Last", email = "first.last@example.com", role = c("aut", "cre"))
Description: What the package does (one paragraph).
Depends: R (>= 3.3.1)
License: What license is it under?
Encoding: UTF-8
LazyData: true

list.files("AdvancedRPkg")
[1] "DESCRIPTION" "NAMESPACE"   "R"           "README.md"
```

The output also shows some of the fields and the information used to fill them in. Since most of these are placeholders, we want to open the files by using a text editor (any should be fine, including RStudio) and modify them. From the directory, `setup()` guesses the package name, but the rest need to be filled out. Using the editor, we change those fields to the following:

```
Package: AdvancedRPkg
Title: An Example R Package for the Book Advanced R
Version: 0.0.0.9000
Authors@R: c(
  person("Matt", "Wiley", email = "matt@elkhartgroup.com", role = c("aut")),
  person("Joshua F.", "Wiley", email = "josh@elkhartgroup.com", role = c("aut", "cre")),
  person("Elkhart Group Ltd.", role = "cph")
  )
Description: This package will demonstrate the basics of an R
  package including documentation and tests.
Depends: R (>= 3.3.1)
License: GPL (>= 3)
Encoding: UTF-8
LazyData: true
```

Note the version number, which is designed in the format `Major.Minor.Patch.DevelopmentVersion`. A good overview of the considerations in determining software versions is described at the Semantic Versioning specification at http://semver.org/. It is quite prescriptive in its recommendations, but it is often helpful to have a fixed set of rules in place for determining a major or minor version of the software. The final piece, the `DevelopmentVersion`, is present only in development versions and is dropped for release. Despite these rules and guides, the reality of many R packages is far more heterogeneous and inconsistent (not everyone agrees on these rules, and even those who agree do not always strictly follow them).

Next up are the authors, described using the `person()` function. In addition to each individual's name and e-mail, three-letter abbreviations are used to describe roles and relations. All possibilities are outlined in the MARC Code List for Relators at www.loc.gov/marc/relators/relaterm.html. However, the most commonly used ones are `aut` for the author, `cre` for a creator who is the person responsible for the project (or in R terms, the person to complain to if there are problems—the maintainer), and `cph` for the copyright holder. Then we provide a few sentences for a description of the package, specify the R version our package depends on, the license, and that data can be loaded on demand (saving startup time).

Adding R Code

With the basics of a package in place, it is time to start adding some R code, the whole point of a package! Since the focus of this chapter is on packaging the code, we reuse some functions and classes from previous chapters. As a first step, we create two files located in the `AdvancedRPkg/R/` directory: `plot_functions.R` and `textplot.R`.

Next, we copy the final `meanPlot()` function from Chapter 4 to make a plot with means. We add the code for this function along with the S3 method, `ggplot.lm()`, that we wrote in Chapter 5 and put both in `plot_functions.R`. Then, we copy the classes and methods (show and subset, which is the bracket operator, `[]`) for the `textplot` class from Chapter 5 into `textplot.R`. Because it is easy to copy the wrong code, we suggest copying and pasting the code from our GitHub repository (https://github.com/ElkhartGroup/AdvancedRPkg). If everything works, it should look like this:

```
list.files("AdvancedRPkg/R")
[1] "plot_functions.R" "textplot.R"
```

As a side note, it can be difficult to decide how many separate files to have. It does not matter for R, but it does make a difference for development. It is not good to have all your code in one file, nor to split every function, class, and method into a separate file. If the result is not too long, we try to group related functions (such as plotting functions) and to group related classes and methods. As common sense, even if you can, do not give files the same names differentiated only by use of lowercase or uppercase letters; it is also generally a bad idea to use special characters or symbols in file names.

Now that we have a basic package template, we can load the functions by using the `load_all()` function from `devtools`. All we have to do is specify the path to the package. We can now see that some of our functions are available in the package namespace, which we do by using the `ls()` function and specifying where we want it to list available objects:

```
load_all("AdvancedRPkg")
Loading AdvancedRPkg
ls(name = "package:AdvancedRPkg")
[1] "ggplot.lm" "meanPlot"  "textplot"
```

In Chapter 4, we briefly discussed scoping. Scoping becomes more important to understand when writing packages. Package authors can write functions with the same names as functions in other packages. Although these functions do not overwrite each other, it can be confusing to be clear about which function you intend to call. This is where the package namespace can be helpful. The *namespace* controls which functions from other packages are imported (and therefore used by code from within that package), and which functions from a package are exported so that they are publicly available to users of the package. You can import specific functions from other packages, using the import feature. If you want many functions from another package, you may decide to make your package depend on that other package, in which case all public functions are available to your package. Another difference between importing and depending on another package is what happens when your package is loaded. If package B depends on package A when a user calls `library(B)`, package B and package A are both loaded and attached. If package B imports from package A, when a user calls `library(B)`, package B gets loaded and attached, and package A is loaded but not attached. Because package A is loaded, its functions are available to code within package B, but they are not exposed directly to the user because package A was not attached. Even when package A (or package B) are loaded and attached, only the exported functions are publicly available to users. This is beneficial, as it allows package developers to write and document functions that are for internal use only. If you do not need to export a function, it is a good idea not to. If two R packages export functions of the same name, and a user loads and attaches both packages, the function from whichever package is loaded later masks the earlier one. For a user, the only choice then is to either not load and attach one package, or to be explicit with the function calls, using the double colon operator, `PkgA::foo()`, `PkgB::foo()`. With thousands of R packages and even more functions, exporting only necessary functions helps avoid such conflicts and masking of names.

All of this is controlled via the `NAMESPACE` file of a package. Although we did not look at it, our package does have a `NAMESPACE` file, which was created when we ran `setup()`. However, the `load_all()` function is unique in that by default when it loads a package, it exports all objects. This is because during development, it is often convenient to be able to call all functions whether they are exported or not.

Tests

With functions and methods added to our new package and working in R, we can begin to think about quality control. It is great to write code that runs, but it is also crucial that the code does what is intended. Now, what is intended and expected are not always the same (that is where documentation plays a critical role, which we cover next). Still, even for the developer, tests ensure that what is written does what it is supposed to do.

Tests may be written at any stage of package development. If functions, names, and features are radically changing, perhaps it is too early to start writing many tests. However, even if a package is not done, if some functions are relatively stable, writing tests early can be helpful. Writing tests earlier rather than later is helpful because it is often easier to write a test immediately after writing a function, when its purpose and the way it works are still fresh in your memory. (If you do not know or have forgotten how a function works, it is hard to test it adequately!). It is also helpful because if later functions build on earlier functions, testing along the way can help ensure that problems are due to the newly written functions and not to some previous building block.

Benefits aside, the practicalities of writing tests are made easier by the `testthat` package. First, we need to create a new subdirectory in our package called `tests`, located at `AdvancedRPkg/tests/`. Next, we create a second subdirectory inside the first named `testthat` (that is,, `AdvancedRPkg/tests/testthat/`), into which we create R source files that run a variety of tests. Note there are other ways to test a package. Here we focus on doing so by using the `testthat` package paradigm.

It is rather difficult to test graphing functions properly, so for our tests, we check whether the subset method for `textplot` works as intended. We can make a file called `test_textplot.R` located under `AdvancedRPkg/tests/testthat/`. We begin with a call to `context()`, which indicates that the tests that follow test related functionality, and we provide the overall name for a suite of tests, `textplot`. Next we set up a simple `textplot` class object, and then run a series of tests. The tests have two components. First, an outer call to `test_that()`. The first argument is a description of the test, and the second is code to do the tests. The code can consist of anything, but commonly includes calls to one of the `expect_*()` functions, of which there are many. We use the `apropos()` function call to show all the options for `expect_*()` before we make our choices:

```
apropos("expect_")
 [1] "expect_cpp_tests_pass"     "expect_equal"
 [3] "expect_equal_to_reference" "expect_equivalent"
 [5] "expect_error"              "expect_failure"
 [7] "expect_false"              "expect_gt"
 [9] "expect_gte"                "expect_identical"
[11] "expect_is"                 "expect_length"
[13] "expect_less_than"          "expect_lt"
[15] "expect_lte"                "expect_match"
[17] "expect_message"            "expect_more_than"
[19] "expect_named"              "expect_null"
[21] "expect_output"             "expect_output_file"
[23] "expect_s3_class"           "expect_s4_class"
[25] "expect_silent"             "expect_success"
[27] "expect_that"               "expect_true"
[29] "expect_type"               "expect_warning"
```

We choose `expect_is()` to check the class of the object and `expect_equal()` to check other features. It is also possible to check that your error checking is working, using `expect_error()` or `expect_warning()`, which "pass" if the code creates an error. We place the following code into our `test_textplot.R` file:

```
context("textplot")

dat <- textplot(
  x = 1:4,
  y = c(1, 3, 5, 2),
  labels = letters[1:4])
```

```
test_that("textplot subset method works with rows only", {
  tmp <- dat[i = 1:2, ]
  expect_is(tmp, "textplot")
  expect_equal(length(tmp@x), 2)
})

test_that("textplot subset method works with variables only", {
  tmp <- dat[, j = c("x", "y")]
  expect_is(tmp, "list")
  expect_equal(length(tmp), 2)
  expect_equal(length(tmp$x), 4)
})

test_that("textplot subset method works with rows and variables", {
  tmp <- dat[1:2, j = c("x", "y")]
  expect_is(tmp, "list")
  expect_equal(length(tmp), 2)
  expect_equal(length(tmp$x), 2)
})
```

We can run the code tests directly. If everything works, there is no output. Output is created only when something does not pass the checks. However, typically, this code is not run directly; it is run in batches. We run all tests by using the `devtools` function, `test()`, and our package name/path. The `devtools` package knows the directory structure for the `testthat` package, and so they play nicely together. To run the tests, `test()` first loads the package by using `load_all()` and then executes the tests. Back in the folder above the package directory, from our chapter-level R file, `chapter06.R`, we run all the tests by executing the code that follows:

```
test("AdvancedRPkg")
Loading AdvancedRPkg
Testing AdvancedRPkg
textplot: ........

DONE ================================================================
```

Errors would be noted, if present. Although we can run the tests as is by using the `devtools` package, for R to run them, we need to add a short file in the `tests` directory (that is, `AdvancedRPkg/tests/`), called `testthat.R`. Into this file, we just need to add a few lines of code, shown next. We do not run this code; this is run by R:

```
library(testthat)
library(AdvancedRPkg)

test_check("AdvancedRPkg")
```

Now, we can use the `covr` package to check code coverage. The `covr` package (https://github.com/jimhester/covr) checks whether different parts of the code are run when a test is executed. For example, a function that contains if/else statements may have only part of its code executed by one test, unless additional tests are derived to check the other conditions. Again, although not required to develop a package, it is a helpful tool to see how well the current tests cover the package functionality. A high level of coverage (aiming toward 100 percent) is a good way to catch bugs and to ensure that new code development does not break old features and functionality; and all of this can also help to assure users that your package is a good choice.

While we used the `checkpoint` library to allow code reproducibility, the nature of testing precludes such library usage from being effective. Thus, installing both `covr` and `testthat` is required. Back in our top-level `chapter06.R` file, we run the `package_coverage()` function to test how well our tests cover our small package's code. The only argument required is the directory where the package is located. In our case, R's working directory is the parent directory, `AdvancedRPkg/`. If the working directory for your R instance is different, you need to specify the appropriate path to get to the `AdvancedRPkg/` directory on your machine:

```
install.packages(c("covr", "testthat"))
cov <- package_coverage("AdvancedRPkg")
```

The output (shown next) indicates that current testing coverage is about 36 percent, and this is driven by coverage of code from `textplot.R`. It is also possible to get a more detailed analysis, using `as.data.frame(cov)`. The output is substantial, so we show just a few rows and columns, but it is an invaluable resource if you think you have tests covering everything and need to figure out what aspects of your code are not being tested yet:

```
cov
AdvancedRPkg Coverage: 35.85%
R/plot_functions.R: 0.00%
R/textplot.R: 52.78%

as.data.frame(cov)[1:3, c(1, 2, 3, 11)]
            filename functions first_line value
1 R/plot_functions.R  meanPlot          2     0
2 R/plot_functions.R  meanPlot          3     0
3 R/plot_functions.R  meanPlot          4     0
```

To have the tests run, we need to indicate that our package requires the `testthat` package. The core functionality of the package does not depend on `testthat`, so instead we add it to the `DESCRIPTION` file under a new section, `Suggests`. Again, using any text editor, we revise `DESCRIPTION` to the following:

```
Package: AdvancedRPkgPackagetests
Title: An Example R Package for the Book Advanced R
Version: 0.0.0.9000
Authors@R: c(
  person("Matt", "Wiley", email = "matt@elkhartgroup.com", role = c("aut")),
  person("Joshua F.", "Wiley", email = "josh@elkhartgroup.com", role = c("aut", "cre")),
  person("Elkhart Group Ltd.", role = "cph")
  )
Description: This package will demonstrate the basics of an R
  package including documentation and tests.
Depends: R (>= 3.3.1)
Suggests:
  testthat
License: GPL (>= 3)
Encoding: UTF-8
LazyData: true
```

Finally, with all the changes we have made to our package, it is time to check in with version control. It is possible to do this from the command line, but we use the GitHub desktop client. By default, all changed files are selected, but rather than commit all changes at once, we do them in related batches of files, along with meaningful messages. The commit messages are added along with each commit and serve to remind you or to let others know a high-level summary of what changes were made or why. It is not necessary to detail every change, as Git takes care of tracking exactly which files changed and how their contents changed.

To begin, we select the `DESCRIPTION`, `NAMESPACE`, and `.gitignore` files and add the message: **initiate R package with DESCRIPTION and NAMESPACE**. The process is shown in Figure 6-5.

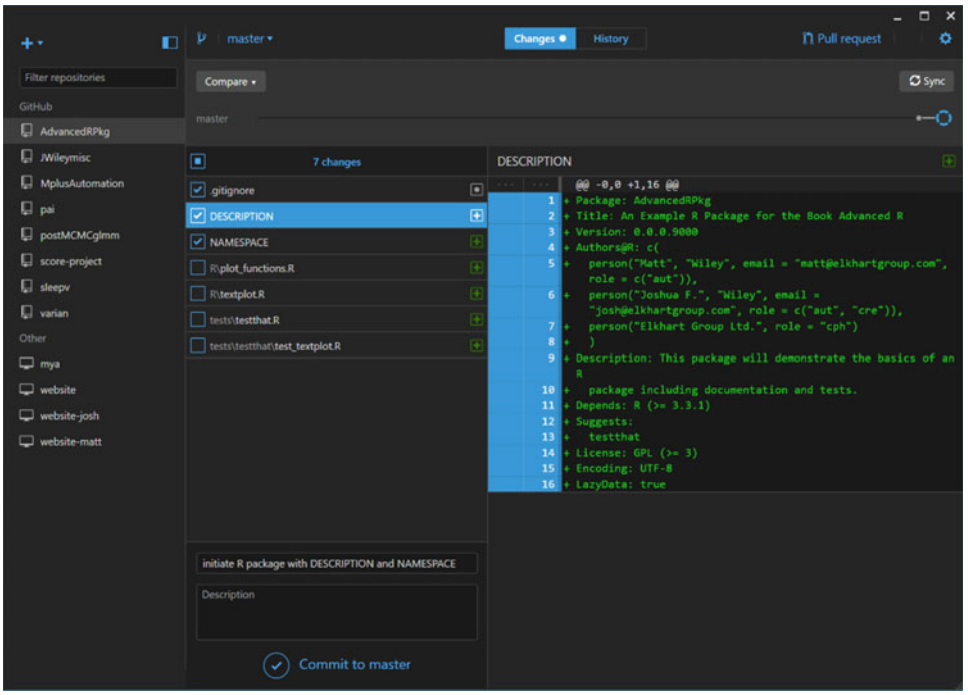


Figure 6-5. *GitHub desktop client selecting changed files, adding a commit message, and committing to the master branch*

Next, we add `plot_functions.R` and `textplot.R` and add the commit message: **initial commit of R functions**. Finally, we add `testthat.R` and `test_textplot.R` and add the commit message: **adding testing for quality control**. Of course, you also could have committed files along the way as they were created. If we switch over to the History tab of the GitHub desktop client, we can see the changes made. If we are happy for them to go public, we click the Sync button, which pulls changes from the GitHub repository to the local repository, and pushes all committed changes from the local repository to the GitHub repository. Note that before you sync, all changes should be committed, because what gets synced are not the actual files in the directory but the Git repositories. Changes to files get added to the local Git repository only when they are committed, so if you do not commit, the updates are not added to the local repository and so do not get pushed to the remote (GitHub) repository. Figure 6-6 shows what this looks like.

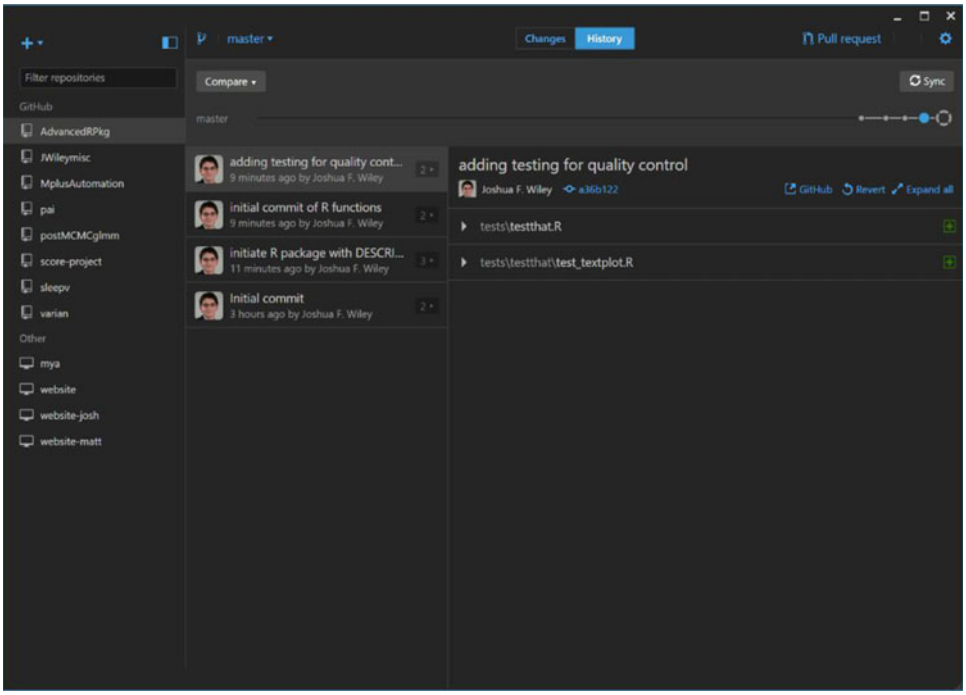


Figure 6-6. *GitHub desktop client showing the history of commits and the Sync button at the top right*

We have only scratched the surface of testing, and obviously our package has a long way to go before it is thoroughly tested. However, these tools should get you started on the right track (and ahead of the majority of the R packages on CRAN) regarding testing and quality control. The next crucial piece is the documentation, a topic we turn to next.

Documentation Using roxygen2

R documentation for functions, data, classes, and methods is located in the `man` subdirectory of a package, and is done by using special R documentation files with the extension `.Rd`. However, for the programmer, this can be harder because the documentation is separated from the actual R code. There is also quite a bit of markup that must be written that is just part of the standard template. The `roxygen2` package, fully automates this process, including the subdirectory creation, and provides an easier way to write documentation by using specially formatted comments next to the R code. Running the `roxygen2` package then converts these particular comments into appropriately formatted `.Rd` files in the `man` directory, so that R has everything it needs to create the help files for the package functions. In this section, we are going to look at how to document several types of objects including functions, data, classes, and methods.

Functions

To document objects using `roxygen2`, documentation is added after a special comment lead: `##'` or `#'`. Because # is R's comment, R ignores it, but the special single quote indicates to `roxygen2` that this information should be processed into the documentation. Note that this syntax precludes commenting this code. All of this code is added to `plot_functions.R` above the `meanPlot()` function code.

As stated, comments are precluded because of syntax; thus the order becomes vital. To see which part maps to which formal, just keep in mind that there are three sections simply separated by line breaks: `title`, `description`, and `details`. The `details` section is optional. We add spaces between line breaks for reader clarity. Function arguments are indicated by using `@param argument_name brief description of the argument`. The argument names listed here must exactly match the named arguments in the function. The `@return` section is where we can indicate what sort of object is returned by the function. In this case, the function is called for the side effect of producing a plot, and the value it returns is not important. Any text can come after `@author` to indicate who wrote the function. Typically, the `@author` section is not required if the author is the same as the overall package author. For the function to be publicly available, it must be exported. This is accomplished by using the directive `@export`. The `roxygen2` package translates this into additional lines of code in the `NAMESPACE` file, indicating it should be exported. The `@keywords` section is optional. Finally, one of the most useful sections for your readers is the `@examples` section, which gives readers executable examples and is one of the easiest ways to show how to use a function.

```
##' Function to plot data and mean summary
##'

##' This is a simple function designed to facilitate plotting raw
##' data along with dots indicating the mean at each x-axis value.
##'

##' Although this function can be used with any type of data that works
##' with \code{plot}, it works best when the x-axis values are discrete,
##' so that there are several y-values at the same x-axis value so that
##' the mean of multiple values is taken.
##'

##' @param formula A formula specifying the variable to be used on the
##'   y-axis and the variable to be used on the x-axis.
##' @param d A data.frame class object containing the variables specified
##'   in the \code{formula}.

##' @return Called for the side effect of creating a plot.
##' @author Wiley
##' @export
##' @keywords plot
##' @examples
##' # example usage of meanPlot
##' meanPlot(mpg ~ factor(cyl), d = mtcars)
```

Now to make the documentation, we can run the `roxygen2` package. This can be done directly by using the `roxygenize()` function and giving it the path to the directory of our package, or it can be done by using the `document()` function from the `devtools` package. We specify the package directory, and the rest happens automatically. The output shows that a documentation file was written (`meanPlot.Rd`) and the `NAMESPACE` file was written:

```
document("AdvancedRPkg")
Updating AdvancedRPkg documentation
Loading AdvancedRPkg
Updating roxygen version in  ~\RFiles\AdvancedRPkg/DESCRIPTION
Writing NAMESPACE
Writing meanPlot.Rd
```

When built or installed, the R documentation file is converted to HTML and PDF. However, already you can preview the development version of the documentation. From the R console, the usual way of getting help for a function should now work:

```
?meanPlot
```

If you use RStudio, you can also open the file and preview it. The resulting HTML file (after building the package) is shown in Figure 6-7. Because of the simplicity of this function and also for the sake of space, we wrote fairly minimal documentation. Often, it is helpful for users and future reference to document more extensively and to carefully explain what each argument can and cannot take. If functions implement new procedures or statistical methods, it is also common to include some references by using the `@references` section.

meanPlot {AdvancedRPkg} R Documentation

Function to plot data and mean summary

Description

This is a simple function designed to facilitate plotting raw data along with dots indicating the mean at each x-axis value.

Usage

```
meanPlot(formula, d)
```

Arguments

`formula`
: A formula specifying the variable to be used on the y-axis and the variable to be used on the x-axis.

`d`
: A data.frame class object containing the variables specified in the `formula`.

Details

Although this function can be used with any type of data that works with `plot`, it works best when the x axis values are discrete, so that there are several y-values at the same x-axis value so that the mean of multiple values is taken.

Value

Called for the side effect of creating a plot.

Author(s)

Wiley

Examples

```
# example usage of meanPlot
meanPlot(mpg ~ factor(cyl), d = mtcars)
```

[Package *AdvancedRPkg* version 0.0.0.9000 Index]

Figure 6-7. *HTML output of the R documentation file for the meanPlot() function*

Data

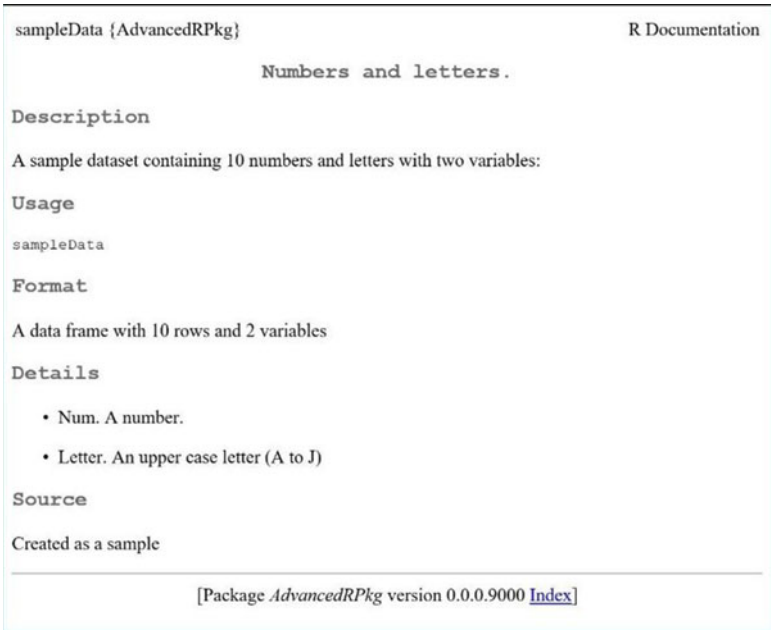
Before we can document data, we need to add some to our package. First, we make a `data` subdirectory in our package folder, located at `AdvancedRPkg/data/`. Then we make a small sample data frame and save it as an `.rda` file by using the code that follows in our `chapter06.R` file. Note again that the file needs to give the correct path to `AdvacedRPkg/data/` based on R's current working directory:

```
sampleData <- data.frame(
  Num = 1:10,
  Letter = LETTERS[1:10])
save(sampleData, file = "AdvancedRPkg/data/sampleData.rda")
```

We add the following code to document the data in a new file called `sampledata.R` located in the `R` subdirectory (that is, `AdvancedRPkg/R/sampledata.R`). This provides a title and details on the data, along with the special parameters `@format` to indicate the format or type of data and `@source` to indicate where it is from. In quotes at the end, the name of the object is given.

```
##' Numbers and letters.
##'
##' A sample data set containing 10 numbers and letters with two variables:
##'
##' \itemize{
##'   \item Num. A number.
##'   \item Letter. An upper case letter (A to J)
##' }
##'
##' @format A data frame with 10 rows and 2 variables
##' @source Created as a sample
"sampleData"
```

After re-roxygenizing the package by using the `document()` function, the resulting HTML is shown in Figure 6-8, again by using the help utilities from the fully built package or by using preview from RStudio.


sampleData {AdvancedRPkg} R Documentation

Numbers and letters.

Description

A sample dataset containing 10 numbers and letters with two variables:

Usage

sampleData

Format

A data frame with 10 rows and 2 variables

Details

- Num. A number.
- Letter. An upper case letter (A to J)

Source

Created as a sample

[Package *AdvancedRPkg* version 0.0.0.9000 Index]

Figure 6-8. *HTML output of the R documentation file for the sampleData data set*

Classes

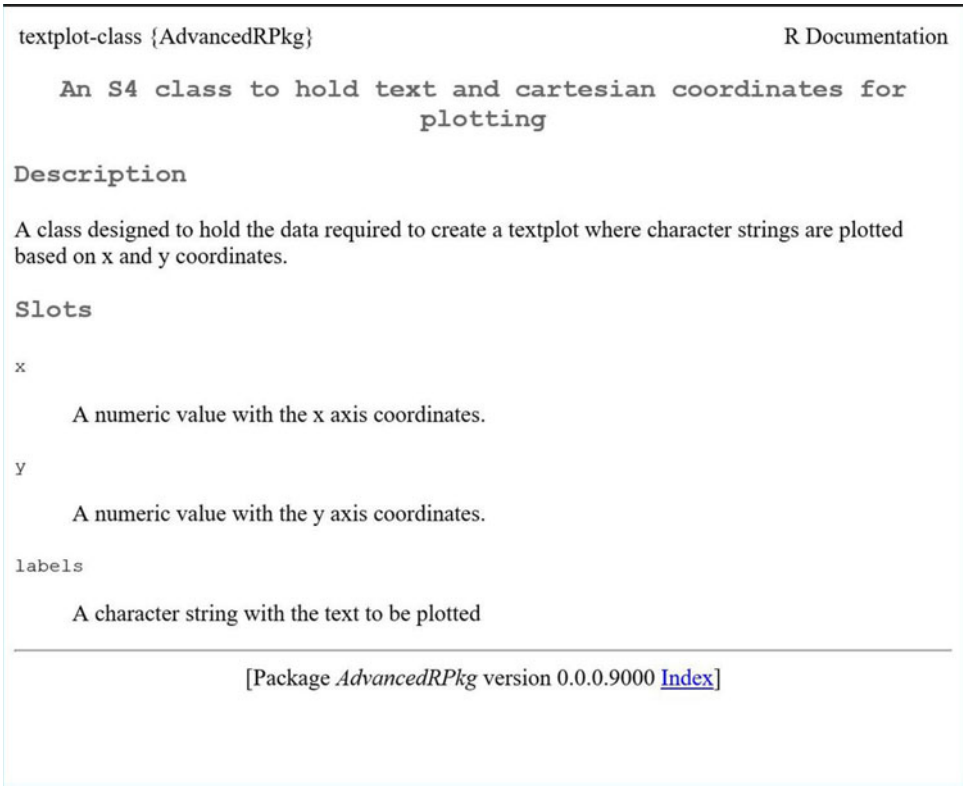
S3 classes are not formal and are not typically documented. S4 classes are straightforward to document. Their documentation goes immediately above the call to `setClass()` in our `textplot.R` file (`AdvancedRPkg/R/textplot.R`) with a title, a details section, and then some parameters— one `@slot` for each slot name, detailing the name and function of the slot. To use the S4 system in a package, we also need to add the `methods` package, which we do by adding `@import` **`methods`** and also adding it to the `Imports` field of the `DESCRIPTION` file. The updated `DESCRIPTION` file with the new `Imports` field is shown here; typically, the `Imports` section immediately follows the `Depends` section:

```
Imports:
  methods
```

The full documentation for the class is shown in the following code:

```
##' An S4 class to hold text and Cartesian coordinates for plotting
##'
##' A class designed to hold the data required to create a textplot
##' where character strings are plotted based on x and y coordinates.
##'
##' @slot x A numeric value with the x axis coordinates.
##' @slot y A numeric value with the y axis coordinates.
##' @slot labels A character string with the text to be plotted
##' @import methods
```

After re-roxygenizing the package by using the `document()` function, the resulting HTML is shown in Figure 6-9.



textplot-class {AdvancedRPkg} R Documentation

An S4 class to hold text and cartesian coordinates for plotting

Description

A class designed to hold the data required to create a textplot where character strings are plotted based on x and y coordinates.

Slots

x

A numeric value with the x axis coordinates.

y

A numeric value with the y axis coordinates.

labels

A character string with the text to be plotted

[Package *AdvancedRPkg* version 0.0.0.9000 Index]

Figure 6-9. *HTML output of the R documentation file for the S4 class textplot*

Methods

Documenting methods is similar to documenting regular functions. S3 methods can be undocumented or documented as a function. The following is the `roxygen2`-style documentation added for the `ggplot.lm` method. The code is added immediately above the function definition in `AdvancedRPkg/R/plot_functions.R`.

Note that here we import the ggplot2 package so that it is available. We also add the ggplot2 package under the Imports section in the DESCRIPTION file, separated by a comma from the R version on which the package depends. The new DESCRIPTION file Depends section is shown here:

```
Imports:
  methods,
  ggplot2
```

By importing it, roxygen2 will automatically add the appropriate import codes to the NAMESPACE file:

```
##' Method for plotting linear models
##'
##' Simple method to plot a linear model using ggplot
##' along with 95% confidence intervals.
##'
##' @param data The linear model object from \code{lm}
##' @param mapping Regular mapping, see \code{ggplot} and \code{aes} for details.
##' @param vars A list of variable values used for prediction.
##' @param \ldots Additional arguments passed to \code{ggplot}
##' @return A ggplot class object.
##' @export
##' @import ggplot2
##' @examples
##' ggplot(
##'   lm(mpg ~ hp * qsec, data = mtcars),
##'   aes(hp, mpg, linetype = factor(qsec)),
##'   vars = list(
##'     hp = min(mtcars$hp):max(mtcars$hp),
##'     qsec = round(mean(mtcars$qsec) + c(-1, 1) * sd(mtcars$qsec)), 1)) +
##'   geom_ribbon(aes(ymin = LL, ymax = UL), alpha = .2) +
##'   geom_line() +
##'   theme_bw()
```

S4 methods can be documented alone, documented with the generic function, or documented with the class. This can be accomplished by using the @describeIn parameter, which is used in place of the title. For our S4 methods, we document them along with the class. The roxygen2 code is relatively brief. We could add details but do not need to. The roxygen2 package takes care of registering the methods, so all we really need to do is add any special notes and document use of parameters. The documentation for the show() method is as follows:

```
##' @describeIn textplot show method
##'
##' @param object The object to be shown
```

We follow this with the documentation for the [operator method. Note that we also export the [method and add an alias, which is just another way that users can look up the method:

```
##' @describeIn textplot extract method
##'
##' @param x the object to subset
##' @param i the rows to subset (optional)
```

```
##' @param j the columns to subset (optional)
##' @param drop should be missing
##' @export
##' @aliases [,textplot-method
```

After re-roxygenizing by using `document()`, the updated HTML help file is shown in Figure 6-10, including the class and additional methods documentation.

textplot-class {AdvancedRPkg} R Documentation

An S4 class to hold text and cartesian coordinates for plotting

Description

A class designed to hold the data required to create a textplot where character strings are plotted based on x and y coordinates.

Usage

```
## S4 method for signature 'textplot'
show(object)

## S4 method for signature 'textplot,ANY,ANY,ANY'
x[i, j, drop = "missing"]
```

Arguments

object	The object to be shown
x	the object to subset
i	the rows to subset (optional)
j	the columns to subset (optional)
drop	should be missing

Methods (by generic)

- show: show method
- [: extract method

Slots

x	A numeric value with the x axis coordinates.
y	A numeric value with the y axis coordinates.
labels	A character string with the text to be plotted

[Package *AdvancedRPkg* version 0.0.0.9000 Index]

Figure 6-10. *HTML output of the R documentation file for the S4 class textplot with added methods documentation*

Building, Installing, and Distributing an R Package

Much as with Git, when building an R package, it can be helpful to ignore some of the files in the directory. We create a file called `.Rbuildignore` containing the following code in our main directory `AdvancedRPkg/`. This file should be located at the same level as the `DESCRIPTION` file:

```
desktop.ini
.Rhistory
.RData
```

After ensuring that the documentation is up-to-date by running `document("AdvancedRPkg")` again, we are almost set. At this point, we have made some changes, so it would be a good idea to add and commit the changes to the Git repository (if you opted into that at the beginning). Although you could add all files at once by running `git add`, it is more informative and easier to revert later (if needed) if changes are committed in chunks based on similar topics. For example, we used separate commits for each of the following, which are shown in the code that follows in chunks:

- The updated code files (every file in the `AdvancedRPkg/R/` directory), `DESCRIPTION`, and `NAMESPACE`, with the commit message *added roxygen style documentation*
- The sample data, with the commit message *added sample data*
- The updated documentation files (every file in the `AdvancedRPkg/man/` directory), with commit message *re-roxygenized package*
- `.Rbuildignore` file, with the commit message *adding file to ignore files during package build*

Now we can build the package into a compressed source tar ball by using the `build()` function from the `devtools` package in our `chapter06.R` file. The output shows the build process and that a `.tar.gz` file is produced at the end:

```
build("AdvancedRPkg")
"c:/usr/MRO/MRO-3.3.1/bin/x64/R" --no-site-file --no-environ  \
  --no-save --no-restore --quiet CMD build "~\Apress_AdvancedR\RFiles\AdvancedRPkg"  \
  --no-resave-data --no-manual

* checking for file '~\Apress_AdvancedR\RFiles\AdvancedRPkg/DESCRIPTION' ... OK
* preparing 'AdvancedRPkg':
* checking DESCRIPTION meta-information ... OK
* checking for LF line-endings in source and make files
* checking for empty or unneeded directories
* looking to see if a 'data/datalist' file should be added
* building 'AdvancedRPkg_0.0.0.9000.tar.gz'

[1] "~/Apress_AdvancedR/RFiles/AdvancedRPkg_0.0.0.9000.tar.gz"
```

We could install and use this. However, it can be helpful to run one more set of checks. The `check()` function in the `devtools` package runs `Rcmd check` on the package. The argument `cran = TRUE` uses the tests for CRAN. The following code should be run from our `chapter06.R` file. The output is extensive, so we omit many sections that work as intended, and highlight some of the checks that would indicate problems we should address. Omitted output is indicated by `[. . .]`.

```
check("AdvancedRPkg", cran = TRUE)
Updating AdvancedRPkg documentation
Loading AdvancedRPkg
[. . .]
* using R version 3.3.1 (2016-06-21)
[. . .]
* checking R code for possible problems ... NOTE
ggplot.lm: no visible global function definition for 'formula'
ggplot.lm: no visible global function definition for 'predict'
ggplot.lm: no visible binding for global variable 'fit'
ggplot.lm: no visible global function definition for 'qnorm'
ggplot.lm: no visible binding for global variable 'se.fit'
meanPlot: no visible global function definition for 'plot'
meanPlot: no visible global function definition for 'points'
show,textplot: no visible global function definition for 'head'
Undefined global functions or variables:
  fit formula head plot points predict qnorm se.fit
Consider adding
  importFrom("graphics", "plot", "points")
  importFrom("stats", "formula", "predict", "qnorm")
  importFrom("utils", "head")
to your NAMESPACE file.
[. . .]
* checking examples ... ERROR
Running examples in 'AdvancedRPkg-Ex.R' failed
The error most likely occurred in:

> base::assign(".ptime", proc.time(), pos = "CheckExEnv")
> ### Name: ggplot.lm
> ### Title: Method for plotting linear models
> ### Aliases: ggplot.lm
>
> ### ** Examples
>
> ggplot(
+   lm(mpg ~ hp * qsec, data = mtcars),
+   aes(hp, mpg, linetype = factor(qsec)),
+   vars = list(
+     hp = min(mtcars$hp):max(mtcars$hp),
+     qsec = round(mean(mtcars$qsec) + c(-1, 1) * sd(mtcars$qsec)), 1)) +
+   geom_ribbon(aes(ymin = LL, ymax = UL), alpha = .2) +
+   geom_line() +
+   theme_bw()
Error: could not find function "ggplot"
Execution halted
* checking for unstated dependencies in 'tests' ... OK
* checking tests ...
  Running 'testthat.R'
 OK
* DONE
```

```
Status: 1 ERROR, 1 NOTE
See
  '~ /Temp/RtmpEnhRD4/AdvancedRPkg.Rcheck/00check.log'R Packagecheck() function
for details.

R CMD check results
1 error  | 0 warnings | 1 note
checking examples ... ERROR
Running examples in 'AdvancedRPkg-Ex.R' failed
The error most likely occurred in:

> base::assign(".ptime", proc.time(), pos = "CheckExEnv")
> ### Name: ggplot.lm
> ### Title: Method for plotting linear models
> ### Aliases: ggplot.lm
>
> ### ** Examples
>
> ggplot(
+   lm(mpg ~ hp * qsec, data = mtcars),
+   aes(hp, mpg, linetype = factor(qsec)),
+   vars = list(
+     hp = min(mtcars$hp):max(mtcars$hp),
+     qsec = round(mean(mtcars$qsec) + c(-1, 1) * sd(mtcars$qsec)), 1)) +
+   geom_ribbon(aes(ymin = LL, ymax = UL), alpha = .2) +
+   geom_line() +
+   theme_bw()
Error: could not find function "ggplot"
Execution halted

checking R code for possible problems ... NOTER Packagecheck() function
ggplot.lm: no visible global function definition for 'formula'
ggplot.lm: no visible global function definition for 'predict'
ggplot.lm: no visible binding for global variable 'fit'
ggplot.lm: no visible global function definition for 'qnorm'
ggplot.lm: no visible binding for global variable 'se.fit'
meanPlot: no visible global function definition for 'plot'
meanPlot: no visible global function definition for 'points'
show,textplot: no visible global function definition for 'head'
Undefined global functions or variables:
  fit formula head plot points predict qnorm se.fit
Consider adding
  importFrom("graphics", "plot", "points")
  importFrom("stats", "formula", "predict", "qnorm")
  importFrom("utils", "head")
to your NAMESPACE file.
```

The `check()` function includes several steps. It first ensures that the documentation is up-to-date, then builds a source package tar ball, and then runs `Rcmd check` on it. We can see here that quite a few issues are caught by the checks. All of them essentially boil down to not telling R exactly what functions or packages are imported or required. Most issues arise because of differences between how packages and interactive R work. During an interactive session, the `base`, `methods`, `graphics`, `utils`, and `stats` packages

are loaded by default, even without calling `library(stats)`, for example. This means that by default in an interactive session, many functions are available on the search path. In packages, R has become stricter in recent versions and requires required functions outside the `base` package to be explicitly imported. The checks even suggest what we could add to our namespace. We use the `roxygen2` package, so rather than adding those to our namespace, we add them to the R code files. Specifically, to `AdvancedRPkg/R/plot_functions.R`, right after the `@export roxygen2` statements, we add this for `meanPlot()`:

```
##' @importFrom graphics plot points
##' @importFrom stats formula
```

And this for the `ggplot.lm()` method in the same file:

```
##' @importFrom stats predict qnorm
```

To `AdvancedRPkg/R/textplot.R`, right after the `@param object roxygen2` statement, we add this for the `show()` method:

```
##' @importFrom utils head
```

In addition, a similar problem occurs with no visible bindings for some global variables. This comes from the `ggplot.lm()` method; because it uses `within()`, R cannot tell that the variables have been defined, even though they are looked up within a data frame environment that contains them. Although this is a somewhat spurious note, it is good to get rid of all notes. One solution is to add a call to `globalVariables()` into our R files somewhere. At the top of `AdvancedRPkg/R/plot_functions.R` , we add the following:

```
globalVariables(c("fit", "se.fit"))
```

The last issue noted is that `ggplot()` could not be found in one of the examples. Because our package imports the `ggplot2` package, the `ggplot()` function is available to code within our package. However, if a user loads only our package, `ggplot2` functions are not loaded for the user's search path. Examples for functions are run as users, and so the function is not available, and R throws an error. The best path forward here is complex. We could omit the example, but then the documentation is less helpful. We could move `ggplot2` from the `Imports` field of the `DESCRIPTION` field to the `Depends` field. Packages depended on are loaded before loading a package. The downside of this approach is that it forces users of our package to have `ggplot2` loaded. This increases the odds of function masking for users because it forces many packages to be loaded. We could also ensure that the code is not run. Finally, we could explicitly point to the `ggplot2` package, by adding `ggplot2::function()`. This is cumbersome, as we use several `ggplot2` functions in that example, and it needs to be added throughout. However, this ensures that the example works and avoids loading `ggplot2` onto users' search path. Users are, of course, free to load `ggplot2` should they wish by explicitly calling `library(ggplot2)` themselves. In the `AdvancedRPkg/R/plot_functions.R` file, we edit the `roxygen2` code for the example for `ggplot.lm()`, as shown here:

```
##' @examples
##' ggplot2::ggplot(
##'   lm(mpg ~ hp * qsec, data = mtcars),
##'   ggplot2::aes(hp, mpg, linetype = factor(qsec)),
##'   vars = list(
##'     hp = min(mtcars$hp):max(mtcars$hp),
##'     qsec = round(mean(mtcars$qsec) + c(-1, 1) * sd(mtcars$qsec)), 1)) +
##'   ggplot2::geom_ribbon(ggplot2::aes(ymin = LL, ymax = UL), alpha = .2) +
##'   ggplot2::geom_line() +
##'   ggplot2::theme_bw()
```

Next we rerun the `check()` function. If all goes well, we should get a status of OK, as shown here:

```
check("AdvancedRPkg", cran = TRUE)
Updating AdvancedRPkg documentation
[. . .]
* DONE

Status: OK

R CMD check results
0 errors | 0 warnings | 0 notes
```

At this point, we can commit all our changes to the Git repository and sync it with GitHub. We can also install the package. First, open a terminal and navigate to the folder containing our package (not the package directory itself, but the directory containing the package directory). From the terminal, we can run `R CMD INSTALL` at an R-enabled terminal such as the Windows command prompt (this can be any terminal, if you have ensured that when you installed R, it was added to the system path):

```
R CMD INSTALL AdvancedRPkg
```

Now, should you wish, `library(AdvancedRPkg)` works in R:

```
library(AdvancedRPkg)
```

You can also share the compressed tar ball created from the `devtools` package from `build()`. If you put the package on GitHub, it can alternately be installed readily by running the following code, replacing `ElkhartGroup` with your username and running this code at the R console (not the OS terminal!):

```
library(devtools)
install_github("ElkhartGroup/AdvancedRPkg")
```

If you want, you can edit the `README` file for the package. It is not required, but can be a useful reference. It is written using the Markdown markup language. If edited, it also shows up on GitHub. We edit the `README.md` file and add the text and markup that follows. Briefly, Markdown uses various numbers of hashes (#) to indicate header levels: # for level 1, and ## for level 2, for example. Text between asterisks, *, is emphasized (italicized). Brackets, [ ], and parentheses, ( ), are used to add URL links, and triple back ticks (```) are used to show the start and end of a block of code.

````
# AdvancedRPkg

This is a sample R package that acompanies Chapter 6 of *Advanced R: Data Programming and
the Cloud*.
To learn more, check out the [book](http://www.apress.com/9781484220764)

## Installation

You can install and test the package by running:

```r
````
````

```
library(devtools)
install_github("ElkhartGroup/AdvancedRPkg")
```

```

Finally, you can submit your package to CRAN should you wish. The first step is to carefully attend to all of their current policies, located at the CRAN Repository Policy site (https://cran.r-project.org/web/packages/policies.html). Once you have checked that your package complies and have corrected any noncompliance, you can submit it to CRAN at https://cran.r-project.org/submit.html. CRAN is huge, with thousands of packages and a tremendous number of updates daily. It is a free service to the community run by volunteers. Thus, even if some of the requirements are tedious, if you want your package on CRAN, it is only fair to play by their rules and do whatever makes it easiest for them. Otherwise, GitHub is a relatively easy place to host and distribute package source code.

# Summary

This chapter has covered the logistics of developing, testing, documenting, and releasing an R package. Although the process does not necessarily involve complex R code, handling the many aspects and getting all pieces to interact properly can be challenging. The payoff for the work is ease of installation and use for users, along with high-quality documentation and assurances that the code works as intended. If you plan to continue developing packages, useful resources for further reading are the official manual, *Writing R Extensions*(https://cran.r-project.org/doc/manuals/R-exts.html), and documentation for using `roxygen2` (https://cran.r-project.org/web/packages/roxygen2/vignettes/roxygen2.html). The vignettes for `roxygen2` are especially useful for topics not covered in this chapter, such as formatting the documentation and collation order (required when some classes or functions have to be loaded before others in your package). A brief summary of the functions used in this chapter is shown in Table 6-3.

This chapter is also the last chapter focused specifically on R programming and the tools around software development in R. The remainder of this book focuses on using R for data management and applied analysis at an advanced level. Although we utilize many aspects of the R programming you've learned in this section of the book and write many functions, we do not develop any new packages in the upcoming chapters.
```

Table 6-3. *Key Functions Described in This Chapter*

Function	What It Does
`setup()`	Creates a new package (in our case, `AdvancedRPkg`).
`list.files()`	Does what it says—the R equivalent of the Unix or Windows `ls`.
`load_all()`	Before a package is built or installed, `library()` does not work; this simulates that in the meantime.
`context()`	Part of the `testthat` package we used in `test_textplot.R`.
`test_that()`	Part of the `testthat` package we used in `test_textplot.R`.
`expect_*()`	A family of functions useful for testing code. These functions check that output matches a certain expectation, such as the class of output, whether the code returns an error or warning, and many others. For a full list, after loading the `testthat` package, run `apropos("expect_")` at the console.
`apropos()`	Useful to search for partially remembered function calls.
`test()`	Runs the `testthat` tests; run from our `chapter06.R` file.
`package_coverage()`	Calculates how much of the package has been tested; usually, the goal is 100 percent.
`document()`	Used to build the `roxygen2` documentation for an R package.
`build()`	Used to build the R package.
`check()`	Used to check the R package; we used the CRAN option in ours.

CHAPTER 7

Introduction to Data Management Using data.table

We already briefly introduced the `data.table` package. This package is the heart of this chapter, which covers the basics of accessing, editing, and manipulating data under the broad term *data management*. Although not glamorous, data management is a critical first step to data visualization or analysis. Furthermore, the majority of time on a particular analysis project may come from the data management. For example, running a linear model in R can take one line of code, once the data is clean and in the format that the `lm()` function in R expects. Data management can be challenging, because raw data come in all types, shapes, and formats; missing data is common; and you may also have to combine or merge separate data sources. In this chapter, we introduce both mechanical and philosophical techniques to approach data management. All packages used in this chapter are already in our `checkpoint.R` file. Thus you need only source the file to get started.

In this chapter, we use the `data.table` R package (Dowle, Srinivasan, Short, and Lianoglou, 2015). The following code loads the `checkpoint` (Microsoft Corporation, 2016) package to control the exact version of R packages used and then loads the `data.table` package:

```
## load checkpoint and required packages
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(data.table)
options(width = 70) # only 70 characters per line
```

Introduction to data.table

One of the benefits of using data tables, rather than the built-in data frame class objects, is that data tables are more memory efficient and faster to manipulate and modify. The `data.table` package accomplishes this to a large extent by altering data tables in place in memory, whereas with data frames, R typically makes a copy of the data, modifies it, and stores it. Making a full copy happens regardless of whether all columns of the data are being changed or, for example, 1 of 100 columns. Consequently, operations on data frames tend to take more time and use up more memory than comparable operations with data tables. To show the concepts of data management, we work with small data, so memory and processor time are not an issue. However, we highlight the use of data tables, rather than data frames, because data tables scale gracefully to far larger amounts of data than do data frames. There are other benefits of data tables as well, including not requiring all variable names to be quoted, that we show throughout the chapter. In contrast, the major advantage of data frames is that they live in base R, and more people are familiar with working with them (which is helpful, for example, if you share your code).

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_7

Data tables can be quite large. Rather than viewing all your data, it is often helpful to view just the `head()` and the `tail()`. When you show a data table in R by typing its name, the first and last five rows are returned by default. R also defaults to printing seven significant digits, which can make our numerical entries messy. For cleanness, we use the `options()` function to control some of the global options so that only the first and last three rows print (if the data table has more than 20 rows total—otherwise, all are shown); in addition, only two significant digits display for our numerical entries. While we set this, we also introduce *Edgar Anderson's iris data*. This data involves sepal and petal lengths and widths in centimeters of three species of iris flowers.

Before we convert the iris data into a data table, we are going to convert our species' names to a character rather factor class. In the following code, we set our options, force the species' names to characters, and create our data table, `diris`:

```
options(stringsAsFactors = FALSE,
        datatable.print.nrows = 20, ## if over 20 rows
        datatable.print.topn = 3, ## print first and last 3
        digits = 2) ## reduce digits printed by R

iris$Species <- as.character(iris$Species)
diris <- as.data.table(iris)

diris
    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
---
148:         6.5         3.0          5.2         2.0 virginica
149:         6.2         3.4          5.4         2.3 virginica
150:         5.9         3.0          5.1         1.8 virginica
```

There are 150 rows of data comprising our three species: `setosa`, `versicolor`, and `virginica`. Each species has 50 measurement sets. R lives in memory, and for huge data tables, this may cause issues. The `tables()` command shows all the data tables in memory, as well as their sizes. In fact, `tables()` is itself a data table. Notice that the function gives information as to the name of our data table(s), the number of rows, columns, the size in `MB`, and if there is a *key*. We discuss data tables with a key in just a few paragraphs. For now, notice that `1MB` is not likely to give us memory issues:

```
tables()
     NAME  NROW NCOL MB
[1,] diris  150    5  1
     COLS                                                    KEY
[1,] Sepal.Length,Sepal.Width,Petal.Length,Petal.Width,Species
Total: 1MB
```

If we need more information about the types of data in each column, calling the `sapply()` command on a data table of interest along with the `class()` function works. Recall that the *`apply` functions return results from calling the function named in the second formal on the elements from the first formal argument. In our case, `class()` is called on `diris`, and we see what types of values are stored in the columns of `diris`:

```
sapply(diris, class)
Sepal.Length  Sepal.Width Petal.Length  Petal.Width      Species
   "numeric"    "numeric"    "numeric"    "numeric"  "character"
```

Data tables have a powerful conceptual advantage over other data formats. Namely, data tables may be keyed. With a key, it is possible to use binary search. For those new to binary search, imagine needing to find a name in a phone book. It would take quite some time to read every name. However, because we know that the telephone directory sorts alphabetically, we can instantly find a middle-of-the-alphabet name, determine whether our search name belongs before or after that name, and remove half the phone book from our search in the process. Our algorithm may be repeated with the remaining half, leaving only a quarter of the original data to search after just two accesses! More mathematically, while a search through all terms of a data frame would be linear based on the number of rows, *n*, in a data table, it is at worse *log n*. In a few new tables we, the authors, used, doubling our data added only minutes to code runtime rather than doubling runtime. A major win.

■ **Note** In `data.table`, a *key* is an index that may or may not be unique, created from one or more columns in the data. The data is sorted by the key, allowing very fast operations using the key. Example operations include subsetting or filtering, performing a calculation (for example, the mean of a variable) for every unique key value, and merging two or more data sets.

Of course, it does take time to key a data table, so it depends on how often you access your data before making this sort of choice. Another consideration is what to use as the key. A table has only one key, although that key may consist of more than one column. The function `setkey()` takes a data table as the first formal and then takes an unspecified number of column names after that to set a key. To see whether a data table has a key, simply call `haskey()` on the data table you wish to test. Finally, if you want to know what columns built the key, the function `key()` called on the data table provides that information. We show these commands on our data table `diris` in the following code lines:

```
haskey(diris)
[1] FALSE
setkey(diris, Species)
key(diris)
[1] "Species"
haskey(diris)
[1] TRUE
```

Keys often are created based on identification variables. For instance, in research, each participant in a study may be assigned a unique ID. In business cases, every customer may have an ID number. Data tables often are keyed by IDs like these, because often operations are performed by those IDs. For example, when conducting research in a medical setting, you may have two data tables, one containing questionnaires completed by participants and another containing data from medical records. The two tables can be joined by participant ID. In business, every time a customer makes a new purchase, an additional row may be added to a data set containing the purchase amount and customer ID. In addition to knowing individual purchases, however, it may also be helpful to know how much each customer has purchased in total—the sum purchase amount for each customer ID.

It is not required to set a key; the essence of binary sort simply requires a logical order. The function call `order()` does this. It is important to note that data tables have another nice feature. Going back to our phone book analogy, imagine that we want to sort by first name instead of last name. Now, the complicated way to do this would be to line up every person in the phone book and then have them move into the new order. In this example, the people are the data. However, it would be much easier simply to reorganize the phone book—much less data. The telephone directory is a reference to the physical location of the people. We can simply move some of those entries higher up in our reference table, without ever changing anyone's physical address. It is again a faster technique that avoids deleting and resaving data in memory. So too, when we impose order on a data table, it is by reference. Note that this new order unsets our key, since a critical aspect of a key is that the data is sorted by that key:

```
diris <- diris[order(Sepal.Length)]
diris
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
  1:          4.3         3.0          1.1         0.1    setosa
  2:          4.4         2.9          1.4         0.2    setosa
  3:          4.4         3.0          1.3         0.2    setosa
 ---
148:          7.7         3.0          6.1         2.3 virginica
149:          7.7         3.8          6.7         2.2 virginica
150:          7.9         3.8          6.4         2.0 virginica
haskey(diris)
[1] FALSE
```

Alternatively, we may order a data table by using multiple variables and even change from the default increasing order to decreasing order. To use more than one column, simply call `order()` on more than column. Adding - before a column name sorts in decreasing order. The following code sorts `diris` based on increasing sepal length and then decreasing sepal width:

```
diris <- diris[order(Sepal.Length, -Sepal.Width)]
diris
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
  1:          4.3         3.0          1.1         0.1    setosa
  2:          4.4         3.2          1.3         0.2    setosa
  3:          4.4         3.0          1.3         0.2    setosa
 ---
148:          7.7         2.8          6.7         2.0 virginica
149:          7.7         2.6          6.9         2.3 virginica
150:          7.9         3.8          6.4         2.0 virginica
```

If we reset our key, it destroys our order. While this would not be apparent with our normal call to `diris`, if we take a look at just the correct rows, we lose the strict increasing order on sepal length:

```
setkey(diris, Species)
diris[49:52]
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1:          5.7         3.8          1.7         0.3     setosa
2:          5.8         4.0          1.2         0.2     setosa
3:          4.9         2.4          3.3         1.0 versicolor
4:          5.0         2.3          3.3         1.0 versicolor
```

What this teaches us is that decisions about order and keys come with a cost. Part of the improved speed of a data table comes with being able to perform fast searches. Choosing a key comes with a one-time computation cost, so it behoves us to make sensible choices for our key. The iris data set is small enough to make such costs irrelevant while we learn, but real-life data sets are not as likely to be so forgiving. Choosing a key also influences some of the default choices for functions we use to understand aspects of our data table. Setting a key is about not only providing a sort of our data, but also asserting our intent to make the key the determining aspect of that data. An example of this is shown with the `anyDuplicated()` function call, which gives the index of the first duplicated row. As we have set our key to `Species`, it defaults to look in just that column. Notice that the first duplicate for `Species` occurs in the second row, while the first duplicated sepal length entry takes place in the third row:

```
anyDuplicated(diris)
[1] 2
anyDuplicated(diris$Sepal.Length)
[1] 3
```

Other functions in the duplication family of functions that are influenced by key include `duplicated()` itself, which returns a Boolean value for each row. In our case, such a call on `diris` would give us `FALSE` values for each of the rows `1`, `51`, and `101`, since we have `50` of each species and are sorting by species. That is many values to get, so we instead put that into a nice table. If we had not set a key, `duplicated()` would consider a row duplicated only if it was a full copy. Contrast the code with our key to an example without a key (we temporarily broke the key for the latter half of the code):

```
haskey(diris)
[1] TRUE
table(duplicated(diris))

FALSE  TRUE
    3   147

##Code With Key Removed
haskey(diris)
[1] FALSE
table(duplicated(diris))

FALSE  TRUE
  149     1
anyDuplicated(diris)
[1] 79
diris[77:80]
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1:          5.8         2.7          3.9         1.2 versicolor
2:          5.8         2.7          5.1         1.9  virginica
3:          5.8         2.7          5.1         1.9  virginica
4:          5.8         2.6          4.0         1.2 versicolor
```

If we return our key to our data set, the last function we discuss shows just those three rows that are `unique()`. If the data set were unkeyed, the unique function would remove only row `79`:

```
unique(diris)
   Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1:          4.3         3.0          1.1         0.1     setosa
2:          4.9         2.4          3.3         1.0 versicolor
3:          4.9         2.5          4.5         1.7  virginica
```

As we have shown in this section, one of the benefits of having a data table is the ability to define a key. The key should be chosen to represent the true uniqueness of a data table. Now, in some types of data, the key might be obvious. Analysis of employee records would likely use employee identity numbers or perhaps social security numbers. However, in other scenarios such as this one, the key might not be so obvious. Indeed, as we see in the preceding results of the `unique()` call on our keyed `diris` data table, we get only three rows returned. That may not truly represent any genuine uniqueness. In that case, it may benefit us to have more than one column used to create the key, although which columns are relevant depends entirely on the data available and the goals of our final analysis. We leave such musings behind for the moment, as we turn our attention to accessing specific parts of our data.

Selecting and Subsetting Data

Using the First Formal

Data tables were built to easily and quickly access data. Part of the way we achieve this is by not having formal row names. Indeed, part of what a key does could be considered providing a useful row name rather than an arbitrary index. The overall format for data table objects is of the form `Data.Table[i, j, by]`, where `i` expects row information. In particular, data tables are somewhat self-aware, in that they understand their column names as variables without using the `$` operator. If a column variable is not named, then it is imagined that we are attempting to call by row, such as in the following bit of code:

■ **Note** Rows from a data table are commonly selected by passing a vector containing one of the following: (1) numbers indicating the rows to choose, `d[1:5]`, (2) negative numbers indicating the rows to exclude, `d[-(1:5)]`, (3) a logical vector `TRUE/FALSE` indicating whether each row should be included, `d[v == 1]`, often as a logical test using a column, `v`, from the data set.

```
diris[1:5]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1:          4.3         3.0          1.1         0.1  setosa
2:          4.4         3.2          1.3         0.2  setosa
3:          4.4         3.0          1.3         0.2  setosa
4:          4.4         2.9          1.4         0.2  setosa
5:          4.5         2.3          1.3         0.3  setosa
```

Notice that if this were a *data frame*, we would have gotten the entire data frame, because we would have been calling for columns and we have only five columns. Here, in *data table* world, because we have not called for a specific column name, we get the first five rows instead. As with increasing or decreasing, we can also perform an opposite selection via the `-` operator. In this case, by asserting that we want the complement of rows `1:148`, we get just the last two rows. Notice the row numbers on the left side. These are rows `149` and `150`, yet, because the data table is not concerned with row names, they are renumbered on the fly as they print to our screen. It is wise to use caution when selecting rows in a data table.

```
diris[-(1:148)]
   Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
1:          7.7         2.6          6.9         2.3 virginica
2:          7.9         3.8          6.4         2.0 virginica
```

More typically, we select by column names or by key. As we have set our key to `Species`, it becomes very easy to select just one species. We simply ask for the character string to match. Now, at the start of this chapter, we made sure that species were character strings. The reason for that choice is clear in the following code, where being able to readily select our desired key values is quite handy. Notice that the second command uses the negation operator, `!`, to assert that we want to select all key elements that are not part of the character string we typed:

```
diris["setosa"]
    Sepal.Length Sepal.Width Petal.Length Petal.Width Species
 1:          4.3         3.0          1.1         0.1  setosa
 2:          4.4         3.2          1.3         0.2  setosa
```

```
 3:          4.4         3.0          1.3         0.2  setosa
---
48:          5.7         4.4          1.5         0.4  setosa
49:          5.7         3.8          1.7         0.3  setosa
50:          5.8         4.0          1.2         0.2  setosa
```

```
diris[!"setosa"]
     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
  1:          4.9         2.4          3.3         1.0 versicolor
  2:          5.0         2.3          3.3         1.0 versicolor
  3:          5.0         2.0          3.5         1.0 versicolor
 ---
 98:          7.7         2.8          6.7         2.0  virginica
 99:          7.7         2.6          6.9         2.3  virginica
100:          7.9         3.8          6.4         2.0  virginica
```

Of course, while we have chosen so far to use key elements, that is by no means required. Logical indexing is quite natural with any of our columns. The overall format compares elements of the named column to the logical tests set up. The final result shows only the subset of rows that evaluated to `TRUE`. For inequalities, multiple arguments, and strict equality (in essence, the usual comparison operators), data tables work as you would expect. Also, note again with data tables, we do not need to reference in which object the variable `Sepal.Length` is. It evaluates within the data table by default.

```
diris[Sepal.Length < 5]
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
 1:          4.3         3.0          1.1         0.1     setosa
 2:          4.4         3.2          1.3         0.2     setosa
 3:          4.4         3.0          1.3         0.2     setosa
---
20:          4.9         3.0          1.4         0.2     setosa
21:          4.9         2.4          3.3         1.0 versicolor
22:          4.9         2.5          4.5         1.7  virginica
```

```
diris[Sepal.Length < 5 & Petal.Width < .2]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1:          4.3         3.0          1.1         0.1  setosa
2:          4.8         3.0          1.4         0.1  setosa
3:          4.9         3.6          1.4         0.1  setosa
4:          4.9         3.1          1.5         0.1  setosa
```

```
diris[Sepal.Length == 4.3 | Sepal.Length == 4.4]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1:          4.3         3.0          1.1         0.1  setosa
2:          4.4         3.2          1.3         0.2  setosa
3:          4.4         3.0          1.3         0.2  setosa
4:          4.4         2.9          1.4         0.2  setosa
```

There are, however, some new operators. In the preceding code, we typed a bit too much because we repeated `Sepal.Length`. Thinking forward to what data tables store, imagine that we have a list of employees that we occasionally modify. We might simply store an employee's unique identifier to that list, and pull from our data table as needed. Of course, in our example here, we have sepal lengths rather than employees, and

we want just the rows that have the lengths of interest in our list. The command is %in% and takes a column name on the left and a vector on the right. Naturally, we may want the complement of such a group, in which case we use the negation operator. We show both results in the following code:

```
interest <- c(4.3, 4.4)
diris[Sepal.Length %in% interest]
   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1:          4.3         3.0          1.1         0.1  setosa
2:          4.4         3.2          1.3         0.2  setosa
3:          4.4         3.0          1.3         0.2  setosa
4:          4.4         2.9          1.4         0.2  setosa
diris[!Sepal.Length %in% interest]
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
  1:          4.5         2.3          1.3         0.3    setosa
  2:          4.6         3.6          1.0         0.2    setosa
  3:          4.6         3.4          1.4         0.3    setosa
 ---
144:          7.7         2.8          6.7         2.0 virginica
145:          7.7         2.6          6.9         2.3 virginica
146:          7.9         3.8          6.4         2.0 virginica
```

Using the Second Formal

Thus far, we have mentioned that data tables do not have native row names, and yet, we have essentially been selecting specific rows in a variety of ways. Recall that our generic layout `Data.Table[i, j, by]` allows us to select objects of columns in the first formal. We turn our attention to the second formal location of `j`, with a goal to select only some columns rather than all columns. Keep in mind, data tables presuppose that variables passed within them exist in the environment of the data table. Thus, it becomes quite easy to ask for all the variables in a single column.

■ **Note** Variables are selected by typing the unquoted variable name, `d[, variable]`, or multiple variables separated by commas within, `.()`, `d[, .(v1, v2)]`.

```
diris[,Sepal.Length]
  [1] 4.3 4.4 4.4 4.4 4.5 4.6 4.6 4.6 4.6 4.7 4.7 4.8 4.8 4.8 4.8 4.8
 [17] 4.9 4.9 4.9 4.9 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.0 5.1 5.1 5.1 5.1
 [33] 5.1 5.1 5.1 5.1 5.2 5.2 5.2 5.3 5.4 5.4 5.4 5.4 5.4 5.5 5.5 5.7
 [49] 5.7 5.8 4.9 5.0 5.0 5.1 5.2 5.4 5.5 5.5 5.5 5.5 5.5 5.6 5.6 5.6
 [65] 5.6 5.6 5.7 5.7 5.7 5.7 5.7 5.8 5.8 5.8 5.9 5.9 6.0 6.0 6.0 6.0
 [81] 6.1 6.1 6.1 6.1 6.2 6.2 6.3 6.3 6.3 6.4 6.4 6.5 6.6 6.6 6.7 6.7
 [97] 6.7 6.8 6.9 7.0 4.9 5.6 5.7 5.8 5.8 5.8 5.9 6.0 6.0 6.1 6.1 6.2
[113] 6.2 6.3 6.3 6.3 6.3 6.3 6.3 6.4 6.4 6.4 6.4 6.4 6.5 6.5 6.5 6.5
[129] 6.7 6.7 6.7 6.7 6.7 6.8 6.8 6.9 6.9 6.9 7.1 7.2 7.2 7.2 7.3 7.4
[145] 7.6 7.7 7.7 7.7 7.7 7.9

is.data.table(diris[,Sepal.Length])
[1] FALSE
```

On the other hand, as you can see in the preceding vector return, entering a single column name does not return a data table. If a data table return is required, this may be solved in more than one way. Recall that the data table strives to be both computationally fast and programmatically fast. We extract single or multiple columns while retaining the data table structure by accessing them in a list. This common desire to pass a `list()` as an argument for data has a clear, shorthand function call in the data table of `.()`. We show in the following code that we may pass a list with one or more elements calling out specific column names. Both return data tables.

```
diris[, .(Sepal.Length)]
     Sepal.Length
  1:          4.3
  2:          4.4
  3:          4.4
 ---
148:          7.7
149:          7.7
150:          7.9

diris[, .(Sepal.Length, Sepal.Width)]
     Sepal.Length Sepal.Width
  1:          4.3         3.0
  2:          4.4         3.2
  3:          4.4         3.0
 ---
148:          7.7         2.8
149:          7.7         2.6
150:          7.9         3.8
```

Using the Second and Third Formals

We have been using the column names as variables that may be directly accessed. They evaluate within the frame of our data table `diris`. This feature may be turned on or off by the `with` argument that is one of our options for the last formal in our generic layout `Data.Table[i, j, by]`. The default value is `with = TRUE`, where default behavior is to evaluate the `j` formal within the frame of our data table `diris`. Changing the value to `with = FALSE` changes the default behavior. This difference is similar to what happens when working with R interactively. We assign the text, **example**, to a variable, `x`. If we type `x` without quotes at the R console, R looks for an object named `x`. If we type `"x"` in quotes, R treats it not as the name of an R object but as a character string.

```
x <- "example"

x
[1] "example"

"x"
[1] "x"
```

When `with = TRUE` (the default), data tables behave like the interactive R console. Unquoted variable names are searched for as variables within the data table. Quoted variable names or numbers are evaluated literally as vectors in their own right. When `with = FALSE`, data tables behave more like data frames, and instead of treating numbers or characters as vectors in their own right, search for the column name or position that matches them. In this context then, our data table expects `j` to be either a column position in numeric form or a character vector of the column name(s). In both cases, data tables are still returned.

```
diris[, 1, with = FALSE]
     Sepal.Length
  1:          4.3
  2:          4.4
  3:          4.4
 ---
148:          7.7
149:          7.7
150:          7.9

diris[, "Sepal.Length", with = FALSE]
     Sepal.Length
  1:          4.3
  2:          4.4
  3:          4.4
 ---
148:          7.7
149:          7.7
150:          7.9
```

This becomes a very useful feature if columns need to be selected dynamically. Character strings pass to data table via variables, such as in the following code:

```
v <- "Sepal.Length"
diris[, v, with = FALSE]
     Sepal.Length
  1:          4.3
  2:          4.4
  3:          4.4
 ---
148:          7.7
149:          7.7
150:          7.9
```

Of course, if we do not turn off the default behavior via the third formal, this may lead to unexpected results. Caution is required to ensure that you get the data you seek. However, remember what each of these return; the results that follow are not, in fact, unfortunate or mistakes. They prove to be quite useful.

```
diris[, v]
[1] "Sepal.Length"
diris[, 1]
[1] 1
```

As you have seen, we may call out one or more columns in a variety of ways, depending on our needs. Recalling the useful notation of - to *subtract* from the data table, we are also able to remove one or more columns from our data table, giving us access to all the rest. Notice in particular that we could use our variable v to hold multiple columns if necessary, thus allowing us to readily keep a list of columns that need removing. For instance, this can be used to remove sensitive information in data that goes from being for internal use only to being available for external. In the following subtraction examples, we ask for the first row in all cases to save space and ink:

```
diris[1, -"Sepal.Length", with=FALSE]
   Sepal.Width Petal.Length Petal.Width Species
1:           3          1.1         0.1  setosa
diris[1, !"Sepal.Length", with=FALSE]
   Sepal.Width Petal.Length Petal.Width Species
1:           3          1.1         0.1  setosa
diris[1, -c("Sepal.Length", "Petal.Width"), with=FALSE]
   Sepal.Width Petal.Length Species
1:           3          1.1  setosa
diris[1, -v, with=FALSE]
   Sepal.Width Petal.Length Petal.Width Species
1:           3          1.1         0.1  setosa
```

Contrastingly, on occasion, we do want to return our column as a vector as with `diris[,Sepal.Length]`. In that case, there are easy-enough ways to gain access to that vector. Here, we combine all our function call examples with the `head()` function to truncate output. Notice that the second example is convenient if you need variable access to that column as a vector, while the last reminds us that data tables build on data frames:

```
head(diris[["Sepal.Length"]]) # easy if you have a R variable
[1] 4.3 4.4 4.4 4.4 4.5 4.6
head(diris[[v]]) # easy if you have a R variable
[1] 4.3 4.4 4.4 4.4 4.5 4.6
head(diris[, Sepal.Length]) # easy to type
[1] 4.3 4.4 4.4 4.4 4.5 4.6
head(diris$Sepal.Length) # easy to type
[1] 4.3 4.4 4.4 4.4 4.5 4.6
```

Variable Renaming and Ordering

Data tables come with variables and a column order. On occasion, it may be desirable to change that order or rename variables. In this section, we show some techniques to do precisely that, keeping our comments brief and letting the code do the talking.

■ **Note** Variable names in data tables can be viewed by using `names(data)`, changed by using `setnames(data, oldnames, newnames)`, and reordered by using `setnames(data, neworder)`.

While we have not yet used this function in this book, `names()` is part of R's base code and is a generic function. Data tables have only column names, so the function call `colnames()` in the `base` package is also going to return the same results:

```
names(diris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width" "Species"
colnames(diris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width" "Species"
```

For objects in the base code of R, such as a data frame, `names()` can even be used to set new values. However, data tables work differently and use a different function to be memory efficient. Remember, data tables avoid copying data whenever possible and instead simply change the reference pointers. Normally, renaming an object in R would copy the data, which both takes time to accomplish and takes up space in memory. The function call for data tables is `setnames()`, which takes formal arguments of the data table, the old column name(s), and the new column name(s):

```
setnames(diris, old = "Sepal.Length", new = "SepalLength")
names(diris)
[1] "SepalLength"  "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

We could have also referred to the old column by numeric position rather than variable name. This can be convenient when names are long, or the data's position is known in advance:

```
setnames(diris, old = 1, new = "SepalL")
names(diris)
[1] "SepalL"       "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

Additionally, columns may be fully reordered. Again, we prefix our command with `set`, as the command `setcolorder()` sets the column order while avoiding any copying of the underlying data. All that changes is the order of the pointers to each column. Thus, this is a fast command for data tables that depends on only the number of columns rather than on the number of elements within a column.

```
setcolorder(diris, c("SepalL", "Petal.Length", "Sepal.Width", "Petal.Width", "Species"))
diris
     SepalL Petal.Length Sepal.Width Petal.Width   Species
  1:    4.3          1.1         3.0         0.1    setosa
  2:    4.4          1.3         3.2         0.2    setosa
  3:    4.4          1.3         3.0         0.2    setosa
 ---
148:    7.7          6.7         2.8         2.0 virginica
149:    7.7          6.9         2.6         2.3 virginica
150:    7.9          6.4         3.8         2.0 virginica
```

As with `setnames()`, `setcolorder()` may be used with numeric positions rather than variable names. We mention here as well that these are all done via reference changes, and our data table `diris` remains keyed throughout all of this.

```
v <- c(1, 3, 2, 4, 5)
setcolorder(diris, v)
diris
     SepalL Sepal.Width Petal.Length Petal.Width   Species
  1:    4.3         3.0          1.1         0.1    setosa
  2:    4.4         3.2          1.3         0.2    setosa
  3:    4.4         3.0          1.3         0.2    setosa
 ---
148:    7.7         2.8          6.7         2.0 virginica
149:    7.7         2.6          6.9         2.3 virginica
150:    7.9         3.8          6.4         2.0 virginica
```

Computing on Data and Creating Variables

We next turn our attention to creating new variables within the data table. Thinking back to a `for` loop structure, the goal would be to step through each row of our data table and make some analysis or change. Data table has very efficient code to do precisely this without formally calling a `for` loop or even using the `*apply()` functions. By *efficient*, we mean faster than either of those two methods in all, or almost all, cases. To get us started on solid ground, we go ahead and re-create and rekey our data table based on the original iris data set to remove any changes we made in the prior sections:

```
diris <- as.data.table(iris)
setkey(diris, Species)
```

■ **Note** New variables are created in a data table by using the `:=` operator in the `j` formal, `data[, newvar := value]`. Variables from the data set can be used as well as using functions.

Creating a new variable in a data table is creating a new column. Because of that, we operate in the second formal, or `j`, area of our generic structure `Data.Table[i, j, by]`. Column creation involves using the assignment by reference `:=` operator. Column creation can take different forms depending on what we wish to create. We may create just a single column and populate it with zeros, create multiple columns with different variables, or even create a new column based on calculations on existing columns. Notice that all these are in the second formal argument location in our data table call, and pay special attention to the way we create multiple columns at once, named `X1` and `X2`, by using `.()` to pass a vector of column names on the left and a list of column values on the right-hand side:

```
diris[, V0 := 0]
diris[, Sepal.Length := NULL]
diris[, c("X1", "X2") := .(1L, 2L)]
diris[, V := Petal.Length * Petal.Width]
diris
     Sepal.Width Petal.Length Petal.Width   Species V0 X1 X2     V
  1:         3.5          1.4         0.2    setosa  0  1  2  0.28
  2:         3.0          1.4         0.2    setosa  0  1  2  0.28
  3:         3.2          1.3         0.2    setosa  0  1  2  0.26
 ---
148:         3.0          5.2         2.0 virginica  0  1  2 10.40
149:         3.4          5.4         2.3 virginica  0  1  2 12.42
150:         3.0          5.1         1.8 virginica  0  1  2  9.18
```

Assignment to `NULL` deletes columns. Full deletion in this way removes any practical ability to restore that column, but it is fast. Another option, of course, would be to use `-` to generate a subset of the original data table and then copy that to a second data table. The difference is in speed and memory usage. Making a copy may almost double your memory usage and takes linear time to copy based on total elements, but making a copy keeps the original around if you want or need to go back. It is possible to delete more than one column at a time via assignment to `NULL`. We show just the first row to verify that the deletion was successful:

```
diris[, c("V", "V0") := NULL]
diris[1]
```

```
   Sepal.Width Petal.Length Petal.Width Species X1 X2
1:         3.5          1.4         0.2  setosa  1  2
```

It is of great benefit to be able to generate new columns based on computations involving current columns. Additionally, this process can be readily modified by commands in the first formal area that allows us to select key data results. Recall that our key is of species, one of which is `setosa`. We re-create our deleted `V` column with the multiplication of two current columns and do so only for rows with the `setosa Species` key. Notice that rows not matching the key get `NA` as filler.

```
diris["setosa", V := Petal.Length * Petal.Width]
unique(diris)
   Sepal.Width Petal.Length Petal.Width    Species X1 X2    V
1:         3.5          1.4         0.2     setosa  1  2 0.28
2:         3.2          4.7         1.4 versicolor  1  2   NA
3:         3.3          6.0         2.5  virginica  1  2   NA
```

However, if we later decide that `virginica` also deserves some values in the `V` column, that can be done, and it need not involve the same calculation as used for `setosa`. We again use the `unique()` function call to look only at the first example of each species. While this example is artificial, imagine a *distance* column being created based on coordinates already stored in the data. Based on key values, perhaps some distance calculations ought to be in Manhattan length, while others might naturally be in Euclidean length.

```
diris["virginica", V := sqrt(Petal.Length * Petal.Width)]
unique(diris)
   Sepal.Width Petal.Length Petal.Width    Species X1 X2    V
1:         3.5          1.4         0.2     setosa  1  2 0.28
2:         3.2          4.7         1.4 versicolor  1  2   NA
3:         3.3          6.0         2.5  virginica  1  2 3.87
```

Such calculations are not limited to column creation. Suppose we want to know the mean of all values in `Sepal.Width`. That calculation can be performed in several ways, each of which may be advantageous depending on your final goal. Recalling that data tables are data frames as well, we could call our arithmetic mean function on a suitable data-frame-style call. However, because we want to find the arithmetic mean of the values of a particular column, we could also call it from the second formal. Remember when we mentioned caution was required for items in this `j` location? It is possible to access values from `j` as vectors. Contrastingly, should we wish to return a data table, we, of course, can. All three of these return the same mean, after all. However, it is the last that may be the most instructive.

```
mean(diris$Sepal.Width)
[1] 3.1
diris[, mean(Sepal.Width)]
[1] 3.1
diris[, .(M = mean(Sepal.Width))]
     M
1: 3.1
```

The last is perhaps best, because it can readily expand. Suppose we want to know the arithmetic mean as in the preceding code, but segregated across our species. This would call for using both the second and third formals. Suppose we also want to know more than just one column's mean. Well, the `.()` notation would allow us to have a convenient way to store that new information in a data table format, thus making it ready to be used for analysis. Notice that we now have the intra-species means, and in the second bit of our code, we have it for more than one column, and in both cases, the returned results are data tables. Thus, we

could access information from these new constructs just as we have been doing all along with `diris`. Much like the hole Alice falls down to reach Wonderland, it becomes a question of "How deep do you want to go?" rather than "How deep can we go?"

```
diris[, .(M = mean(Sepal.Width)), by = Species]
      Species   M
1:     setosa 3.4
2: versicolor 2.8
3:  virginica 3.0
diris[, .(M1 = mean(Sepal.Width), M2 = mean(Petal.Width)), by = Species]
      Species  M1   M2
1:     setosa 3.4 0.25
2: versicolor 2.8 1.33
3:  virginica 3.0 2.03
```

To see the full power of these new data tables we are building, suppose we want to see whether there's any correlation between the means of species' sepal and petal widths. Further imagine that we might find a use for the mean data at some future point beyond the correlation. We can treat the second data table in the preceding code snippet as a data table (because it is), and use the second formal to run correlation. This code follows a `Data.Table[ i, j, by] [ i, j, by]` layout, which sounds rather wordy written out; take a look at the following code to see how it works:

```
diris[, .(M1 = mean(Sepal.Width), M2 = mean(Petal.Width)), by = Species][, .(r = cor(M1,
M2))]
       r
1: -0.76
```

Variables may create variables. In other words, a new variable may be created that indicates whether a petal width is greater or smaller than the median petal width for that particular species. In the code that follows, `median(Petal.Width)` is calculated by `Species`, and the `TRUE` or `FALSE` values are stored in the newly created column `MedPW` for all 150 rows:

```
diris[, MedPW := Petal.Width > median(Petal.Width), by = Species]
diris
     Sepal.Width Petal.Length Petal.Width   Species X1 X2    V MedPW
  1:         3.5          1.4         0.2    setosa  1  2 0.28 FALSE
  2:         3.0          1.4         0.2    setosa  1  2 0.28 FALSE
  3:         3.2          1.3         0.2    setosa  1  2 0.26 FALSE
 ---
148:         3.0          5.2         2.0 virginica  1  2 3.22 FALSE
149:         3.4          5.4         2.3 virginica  1  2 3.52  TRUE
150:         3.0          5.1         1.8 virginica  1  2 3.03 FALSE
```

Just as it was possible to have multiple arguments passed to the second formal, it is possible to have multiple arguments passed to the third via the same `.()` list function call. Again, because what is returned is a data table, we can also treat it as one with the brackets and perform another layer of calculations on our results:

```
diris[, .(M1 = mean(Sepal.Width), M2 = mean(Petal.Width)), by = .(Species, MedPW)]
      Species MedPW  M1   M2
1:     setosa FALSE 3.4 0.19
2:     setosa  TRUE 3.5 0.38
3: versicolor  TRUE 3.0 1.50
```

```
4: versicolor FALSE 2.6 1.19
5:  virginica  TRUE 3.1 2.27
6:  virginica FALSE 2.8 1.81
```

```
diris[, .(M1 = mean(Sepal.Width), M2 = mean(Petal.Width)), by = .(Species, MedPW)]
            [, .(r = cor(M1, M2)), by = MedPW]
   MedPW     r
1: FALSE -0.78
2:  TRUE -0.76
```

Performing computations in the `j` argument is one of the most powerful features of data tables. For example, we have shown relatively simple computations of means, medians, and correlations. However, nearly any function can be used. The only caveat is that if you want the output also to be a data table, the function should return or be manipulated to return output appropriate for a data table. For instance, a linear model object that is a list of varying length elements would be messy to coerce into a data table, as it is not tabular data. However, you could easily extract regression coefficients as a vector, and that could become a data table.

Merging and Reshaping Data

Managing data often involves a desire to combine data from different sources into a single object or to reshape data in one object to a different format. When combining data, there are three distinct types of *data merge*: one-to-one, one-to-many (or many-to-one), and many-to-many. You'll learn about these types in this section. The general process is to look at data from two or more collections and then compare keys, with a goal to successfully connect information by row. This is called *horizontal merging* because it increases the horizontal length of the target data set.

Merging Data

Merges in which each key is unique per data set are of the *one-to-one* form. An example is merging data sets keyed by social security number (SSN) that contain official mailing addresses in one and preferred e-mails in another. In that case, we would expect the real-world data set to have one address per SSN, and the preferred e-mail data set to have only one row per SSN. Contrastingly, a *one-to-many* or *many-to-one* merge might occur if we had our official mailing address data set as well as all known e-mail addresses, where each new e-mail address tied to one SSN had its row. If the goal were to append all e-mails to a single real-world address row, it would be many-to-one, and column names such as `Email001` and `Email002` might collapse to a single row. On the other hand, if the goal were to append a real-world address to each e-mail, then perhaps only one new column would be added, although the data copies to each row of the e-mail address. Finally, there can be *many-to many*-merges, in which multiple rows of data append to multiple rows. Figure 7-1 shows examples of these three types of data merges.

One to One Merge			
Mailing Address		Preferred Email Address	
123-45-6789	123 Fake Street	123-45-6789	123Fake@elkhartgroup.com
321-54-9876	321 Fake Street	321-54-9876	321Fake@elkhartgroup.com

One to Many or Many to One (depending on direction)			
Mailing Address		All Email Addresses	
123-45-6789	123 Fake Street	123-45-6789	123Fake@elkhartgroup.com
321-54-9876	321 Fake Street	123-45-6789	info@elkhartgroup.com
		321-54-9876	321Fake@elkhartgroup.com

Many to Many Merge			
All Known Addresses		All Email Addresses	
123-45-6789	123 Fake Street	123-45-6789	123Fake@elkhartgroup.com
123-45-6789	123 Other Road	123-45-6789	info@elkhartgroup.com
321-54-9876	321 Fake Street	321-54-9876	321Fake@elkhartgroup.com
321-54-9876	321 Other Road		

Figure 7-1. *Some imaginary data showing how various merges might occur*

It is worth noting that the difficulty with merging may well be in understanding what is being done rather than understanding the code itself. For data tables, the function call is `merge()`, and arguments control the type of merge that is performed. Due to the keyed nature of data tables, merging by the key choice can be quite efficient. Choosing to merge by nonkey columns is also possible, but will not be as efficient computationally.

Before getting into the first merge code example, we generate four data tables with information unique to each. All are keyed by `Species` from the iris data set. In real life, such code is not required because normally data tables come already separated. This code is included only for reproducibility of the merge results. Also note that these are very short data sets; the goal here is for them to be viewed and fully understood independently. This provides the best environment from which to understand the final merge results.

```
diris <- diris[, .(Sepal.Width = Sepal.Width[1:3]), keyby = Species]

diris2 <- unique(diris)
```

```
dalt1 <- data.table(
  Species = c("setosa", "setosa", "versicolor", "versicolor",
              "virginica", "virginica", "other", "other"),
  Type = c("wide", "wide", "wide", "wide",
           "narrow", "narrow", "moderate", "moderate"),
  MedPW = c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE))
setkey(dalt1, Species)

dalt2 <- unique(dalt1)[1:2]

diris
      Species Sepal.Width
1:     setosa         3.5
2:     setosa         3.0
3:     setosa         3.2
4: versicolor         3.2
5: versicolor         3.2
6: versicolor         3.1
7:  virginica         3.3
8:  virginica         2.7
9:  virginica         3.0

diris2
      Species Sepal.Width
1:     setosa         3.5
2: versicolor         3.2
3:  virginica         3.3

dalt1
      Species     Type MedPW
1:      other moderate  TRUE
2:      other moderate FALSE
3:     setosa     wide  TRUE
4:     setosa     wide FALSE
5: versicolor     wide  TRUE
6: versicolor     wide FALSE
7:  virginica   narrow  TRUE
8:  virginica   narrow FALSE

dalt2
   Species     Type MedPW
1:   other moderate  TRUE
2:  setosa     wide  TRUE
```

■ **Note** A one-to-one merge requires two keyed data tables, with each row having a unique key:

`merge(x = dataset1, y = dataset2)`. The merge exactly matches a row from one data table to the other, and the result includes all the columns (variables) from both data tables. Whether nonmatching keys are included is controlled by the `all` argument, which defaults to `FALSE` to include only matching keys, but may be set to `TRUE` to include nonmatching keys as well.

In a *one-to-one merge*, there is at most one row per key. In the case of `diris2`, there are three rows, and each has a unique species. Similarly, in `dalt2`, there are two rows, and each has a unique species. The only common key between these two data sets is `setosa`. A `merge()` call on this one-to-one data compares keys and by default creates a new data table that combines only those rows that have matching keys. In this case, merging yields a tiny data table with one row. The original two data tables are not changed in this operation. The call to `merge()` takes a minimum of two arguments; the first is `x`, and the second is `y`, which represent the two data sets to be merged.

```
merge(diris2, dalt2)
   Species Sepal.Width Type MedPW
1:  setosa         3.5 wide  TRUE
```

Notice that what has been created is a complete data set. However, it is quite small, because the only results returned were those for which full and complete matches were possible. Depending on the situation, different results may be desired from a one-to-one merge. Perhaps all rows of `diris2` are wanted regardless of whether they match `dalt2`. Conversely, the opposite may be the case, and all rows of `dalt2` may be desired regardless of whether they match `diris2`. Finally, to get all possible data, it may be desirable to keep all rows that appear in either data set. These objectives may be accomplished by variations of the `all = TRUE` formal argument. In particular, note how to control which single data set is designated as essential to keep via the `.x` or `.y` suffix to `all`. If data is not available for a specific row or variable, the cell is filled with a missing value (`NA`).

```
merge(diris2, dalt2, all.x = TRUE)
      Species Sepal.Width Type MedPW
1:     setosa         3.5 wide  TRUE
2: versicolor         3.2   NA    NA
3:  virginica         3.3   NA    NA
merge(diris2, dalt2, all.y = TRUE)
   Species Sepal.Width     Type MedPW
1:   other          NA moderate  TRUE
2:  setosa         3.5     wide  TRUE
merge(diris2, dalt2, all = TRUE)
      Species Sepal.Width     Type MedPW
1:      other          NA moderate  TRUE
2:     setosa         3.5     wide  TRUE
3: versicolor         3.2       NA    NA
4:  virginica         3.3       NA    NA
```

■ **Note** A one-to-many or many-to-one merge involves one data table in which each row has a unique key and another data table in which multiple rows may share the same key. Rows from the data table with unique keys are repeated as needed to match the data table with repeated keys. In R, identical code is used as for a one-to-one merge: `merge(x = dataset1, y = dataset2)`.

Merges involving `diris` and `dalt2` are of the one-to-many or many-to-one variety. Again, the behavior wanted in the returned results may be controlled when it comes to keeping only full matches or allowing various flavors of incomplete data to persist through the merge. The three examples provided are not exhaustive, yet they are representative of the different possibilities. The last two cases may not be desirable, as `other` has only one row, whereas most species have three.

```
merge(diris, dalt2)
   Species Sepal.Width Type MedPW
1:  setosa         3.5 wide  TRUE
2:  setosa         3.0 wide  TRUE
3:  setosa         3.2 wide  TRUE

merge(diris, dalt2, all.y = TRUE)
   Species Sepal.Width     Type MedPW
1:   other          NA moderate  TRUE
2:  setosa         3.5     wide  TRUE
3:  setosa         3.0     wide  TRUE
4:  setosa         3.2     wide  TRUE

merge(diris, dalt2, all = TRUE)
       Species Sepal.Width     Type MedPW
 1:      other          NA moderate  TRUE
 2:     setosa         3.5     wide  TRUE
 3:     setosa         3.0     wide  TRUE
 4:     setosa         3.2     wide  TRUE
 5: versicolor         3.2       NA    NA
 6: versicolor         3.2       NA    NA
 7: versicolor         3.1       NA    NA
 8:  virginica         3.3       NA    NA
 9:  virginica         2.7       NA    NA
10:  virginica         3.0       NA    NA
```

So far, one-to-one and one-to-many merges have been demonstrated. These types of merges, or joins, have a final, merged number of rows that is at most the sum of the rows of the two data tables. As long as at least one of the dimensions of the merge are capped to one, there is no concern. However, in a *many-to-many* scenario, the upper limit of size becomes the product of the row count of the two data tables. In many research contexts, many-to-many merges are comparatively rare. Thus, the default behavior of a merge for data tables is to prevent such a result, and `allow.cartesian = FALSE` is the mechanism through which this is enforced as a formal argument of `merge()`. The same is not true for other data structures but is unique to data tables.

Attempting to merge `diris` and `dalt1` results in errors from this default setting, which disallows merged table rows to be larger than the total *sum* of the rows. With three `setosa` keys in `diris` and two `setosa` keys in `dalt1`, a merged table would result with `3 x 2 = 6` total rows for `setosa` alone. As many-to-many merges involve a multiplication of the total number of rows, it is easy to end up with a very large database. Allowing this would yield the following results. Notice that this result has only complete rows.

```
merge(diris, dalt1, allow.cartesian = TRUE)
       Species Sepal.Width   Type MedPW
 1:     setosa         3.5   wide  TRUE
 2:     setosa         3.5   wide FALSE
 3:     setosa         3.0   wide  TRUE
 4:     setosa         3.0   wide FALSE
 5:     setosa         3.2   wide  TRUE
 6:     setosa         3.2   wide FALSE
 7: versicolor         3.2   wide  TRUE
 8: versicolor         3.2   wide FALSE
 9: versicolor         3.2   wide  TRUE
10: versicolor         3.2   wide FALSE
11: versicolor         3.1   wide  TRUE
```

```
12: versicolor         3.1   wide FALSE
13:  virginica         3.3 narrow  TRUE
14:  virginica         3.3 narrow FALSE
15:  virginica         2.7 narrow  TRUE
16:  virginica         2.7 narrow FALSE
17:  virginica         3.0 narrow  TRUE
18:  virginica         3.0 narrow FALSE
```

The preceding has only complete rows and is missing the `other` species. Allowing all possible combinations nets two more rows, for a total of 20. It is an entirely different conversation if the resulting data is useful. When merging data, it always pays to consider what types of information are available, and what might be needed for a particular bit of analysis. We suppress rows 4 through 19 here:.

```
merge(diris, dalt1, all = TRUE, allow.cartesian = TRUE)
       Species Sepal.Width     Type MedPW
 1:      other          NA moderate  TRUE
 2:      other          NA moderate FALSE
 3:     setosa         3.5     wide  TRUE
---
20:  virginica         3.0   narrow FALSE
```

For all of these, the key is `Species`, and that is only one column. Furthermore, it is nonunique. It is also possible to have keys based on multiple columns. Recognize that key choice may make all the difference between a one-to-one versus a many-to-many merge. Consider the following instructive example, in which only a single column is used as the key and the data is merged:

```
d2key1 <- data.table(ID1 = c(1, 1, 2, 2), ID2 = c(1, 2, 1, 2),  X = letters[1:4])
d2key2 <- data.table(ID1 = c(1, 1, 2, 2), ID2 = c(1, 2, 1, 2),  Y = LETTERS[1:4])

d2key1
   ID1 ID2 X
1:   1   1 a
2:   1   2 b
3:   2   1 c
4:   2   2 d

d2key2
   ID1 ID2 Y
1:   1   1 A
2:   1   2 B
3:   2   1 C
4:   2   2 D

setkey(d2key1, ID1)
setkey(d2key2, ID1)

merge(d2key1, d2key2)
   ID1 ID2.x X ID2.y Y
1:   1     1 a     1 A
2:   1     1 a     2 B
3:   1     2 b     1 A
```

```
4:   1     2 b     2 B
5:   2     1 c     1 C
6:   2     1 c     2 D
7:   2     2 d     1 C
8:   2     2 d     2 D
```

For the merge to be successful with its duplicated column names, `ID2.x` and `ID2.y` were created. The fact is, this data should have resulted in only four rows, once merged, because the data in those two columns is identical! By setting the key for both data sets to involve both columns `ID1` and `ID2`, not only is the many-to-many merge reduced to a one-to-one merge, but the data is of a more practical nature. Multiple columns being used to key can be especially useful when merging multiple long data sets. For instance, with repeated measures on multiple people over time, there could be an ID column for participants and another time variable, but both would be used to merge properly. We discuss reshaping data next.

```
setkey(d2key1, ID1, ID2)
setkey(d2key2, ID1, ID2)

merge(d2key1, d2key2)
   ID1 ID2 X Y
1:   1   1 a A
2:   1   2 b B
3:   2   1 c C
4:   2   2 d D
```

Reshaping Data

In addition to being merged, data may be reshaped. Consider our `iris` data set: each row represents a unique flower measurement, yet many columns are used for each flower measured. This is an example of *wide data*. Despite not having a unique key, there are a variety of measurements for each flower (row). Wide data may be reshaped to *long data*, in which each key appears multiple times and stores variables in fewer columns. First, we reset our data to the `iris` data set and create a unique ID to be our key based on the row position. Recall that data tables suppress rows, so we add those into their `ID` column by using another useful shorthand of data tables, namely `.N` for number of rows:

```
diris <- as.data.table(iris)
diris[, ID := 1:.N]
setkey(diris, ID)
diris
     Sepal.Length Sepal.Width Petal.Length Petal.Width   Species  ID
  1:          5.1         3.5          1.4         0.2    setosa   1
  2:          4.9         3.0          1.4         0.2    setosa   2
  3:          4.7         3.2          1.3         0.2    setosa   3
 ---
148:          6.5         3.0          5.2         2.0 virginica 148
149:          6.2         3.4          5.4         2.3 virginica 149
150:          5.9         3.0          5.1         1.8 virginica 150
```

■ **Note** Reshaping is used to transform data from wide to long or from long to wide format. In *wide* format, a repeatedly measured variable has separate columns, with the column name indicating the assessment. In *long* format, a repeated variable has only one column, and a second column indicates the assessment or "time." Wide-to-long reshaping is accomplished by using the `melt()` function, which requires several arguments.

To reshape long, we `melt()` the data table measure variables into a stacked format, which is based on the `reshape2` and `reshape` packages (Wickham, 2007), although there is a `data.table` method for `melt()`, which is what is dispatched in this case. This function takes several arguments, which we'll discuss in order. To see all the arguments, examine the help: `?melt.data.table`. Note that it is important to specify the exact method, `melt.data.table`, because the arguments for the generic `melt()` function are different.

The first formal, `data`, is the name of the data table to be melted, followed by `id.vars`, which is where the ID variables and any other variable that should not be reshaped long. By default, `id.vars` is any column name not explicitly called out in the next formal. Any column name assigned to `id.vars` is not reshaped into long form; it is repeated, but left as is. The third formal is for the measure variables, and `measure.vars` uses these column names to stack the data. Typically, this is a list with separate character vectors to indicate which variables should be stacked together in the long data. The fourth formal is `variable.name`, which is the name of the new variable that will be created to indicate which level of the `measure.vars` a specific row in the new long data set belongs to. In the following example, the first level is for length measurements, while the second level corresponds to width measurements of iris flowers. Finally, `value.name` is the name of the *molten* (or long) column(s), which are the names of the columns in our long data containing the original wide variables' values.

Describing reshaping and the required arguments is rather complex. It may be easier to compare output between the now familiar iris data set and the reshaped, long iris data set, shown here:

```
diris.long <- melt(diris, measure.vars =
                   list( c("Sepal.Length", "Sepal.Width"), c("Petal.Length", "Petal.
                   Width")),
              variable.name = "Type", value.name = c("Sepal", "Petal"),
              id.vars = c("ID", "Species"))
diris.long
      ID   Species Type Sepal Petal
  1:   1    setosa    1   5.1   1.4
  2:   2    setosa    1   4.9   1.4
  3:   3    setosa    1   4.7   1.3
 ---
298: 148 virginica    2   3.0   2.0
299: 149 virginica    2   3.4   2.3
300: 150 virginica    2   3.0   1.8
```

Notice that our data now takes 300 rows because of being melted. The `Type` `1` versus `2` may be confusing to equate with length versus width measurements for sepals and petals. The `melt()` method for data tables defaults to just creating an integer value indicating whether it is the first, second, or third (and so forth) level. To make the labels more informative, we use `factor()`. Because factor will operate on a column and change the row elements of the column, it belongs in the second formal `j` argument of the data table.

```
diris.long[, Type := factor(Type, levels = 1:2, labels = c("Length", "Width"))]
diris.long
      ID   Species   Type Sepal Petal
  1:   1    setosa Length   5.1   1.4
  2:   2    setosa Length   4.9   1.4
  3:   3    setosa Length   4.7   1.3
 ---
298: 148 virginica  Width   3.0   2.0
299: 149 virginica  Width   3.4   2.3
300: 150 virginica  Width   3.0   1.8
```

The data set `diris.long` is a good example of a long data set. It is not, however, the longest possible data set from `diris`–not that such a thing is necessarily a goal, by any means. However, should `melt()` be called on just `diris` along with calling out `id.vars`, the resulting data table is perhaps the longest possible form of the `iris` data. Just as `id.vars` defaults to any values not in `measure.vars`, so too `measure.vars` defaults to any values not in `id.vars`. As you can also see, `variable.name` and `value.name` default to `variable` and `value`, respectively. Notice that `diris.long2` has 600 rows, and what were formerly the four columns of sepal and petal lengths and widths are now two columns. Also note that when this is done, `melt()` automatically sets the values of `variable` to the initial variable names from the data set.

```
diris.long2 <- melt(diris, id.vars = c("ID", "Species"))
diris.long2
      ID   Species     variable value
  1:   1    setosa Sepal.Length   5.1
  2:   2    setosa Sepal.Length   4.9
  3:   3    setosa Sepal.Length   4.7
 ---
598: 148 virginica  Petal.Width   2.0
599: 149 virginica  Petal.Width   2.3
600: 150 virginica  Petal.Width   1.8
```

■ **Note** To reshape data from long to wide, the `dcast()` function is used. Two required arguments are `data` and `formula`, which indicate how the data should be reshaped from long to wide. The formula must indicate unique values in a cell, or the data needs to be aggregated within a cell.

While `melt()` is used to make wide data long, `dcast()` is used to make long data wide. Without unique identifiers for data, multiple values must be aggregated to fit in one cell. Thus, calling our casting function without some thought may create unwanted results (although sometimes aggregating multiple values is exactly what is desired). Notice that in the second formal, `Species` is called out as the identity key, and `variable` is named as the location to find our new column names:

```
dcast(diris.long2, Species ~ variable)
Aggregate function missing, defaulting to 'length'
      Species Sepal.Length Sepal.Width Petal.Length Petal.Width
1:     setosa           50          50           50          50
2: versicolor           50          50           50          50
3:  virginica           50          50           50          50
```

If this long data is to be better cast to wide, a unique identifier is needed so that the formula creates unique cells that contain only one value. Unless aggregating results is desired, for a given variable, the `formula` argument should result in a single value, not multiple values.

In this case, since there are four possible values that the variable column takes on in the long data, as long as there is a unique key that is never duplicated over a specific variable such as `Sepal.Length`, the cast works out. In other words, if the wide data has a unique key, we can combine that unique key with the variable names in the long data, which together will exactly map cells from the long data set back to the wide data. The left side of the formula contains the key or ID variable. This side of the formula (left of the tilde, ~) may also contain any other variable that does not vary (such as would have been specified using the `id.vars` argument to `melt()` if the data had initially been reshaped from wide to long). The right-hand side of the formula (after the tilde, ~) contains the variable or variables that indicate which row of the long data set belongs in which column of the new wide data set.

In our long iris data set example, the `ID` and `Species` variables do not vary and so belong on the left side. The variable called `variable` indicates whether a particular value in the long data set represents `Sepal.Length`, `Sepal.Width`, `Petal.Length`, or `Petal.Width`. Thus, for this example, the formula looks like `ID + Species ~ variable`:

```
diris.wide2 <- dcast(diris.long2, ID + Species ~ variable)
diris.wide2
      ID   Species Sepal.Length Sepal.Width Petal.Length Petal.Width
  1:   1    setosa          5.1         3.5          1.4         0.2
  2:   2    setosa          4.9         3.0          1.4         0.2
  3:   3    setosa          4.7         3.2          1.3         0.2
 ---
148: 148 virginica          6.5         3.0          5.2         2.0
149: 149 virginica          6.2         3.4          5.4         2.3
150: 150 virginica          5.9         3.0          5.1         1.8
```

Going back to the first `diris.long` data with the two-level factor for length and width of sepals and petals, it is also possible to cast this back to wide. The call to `dcast()` begins similarly to our previous example. We specify the long data set, a formula indicating the `ID` and `Species` variables that are not stacked, and the variable, `Type`, that indicates what each row of the data set belongs to.

In such not-so-long (for lack of a better word) data, we require an additional argument, the `value.var` formal, to let `dcast()` know which variables in the long data are repeated measures. This can be a character vector of every variable in the long data set that will correspond to multiple variables in the wide data set—in our case, `Sepal` and `Petal`, which in the long data contain both lengths and widths, as indicated in the `Type` variable. Additionally, the separator must be chosen, as data table column names may have different separators. This is selected via the `sep =` formal, and here we use the full stop for continuity's sake, as that is how the variables were labeled in the original wide iris data set. We show merely the `dcast()` command, and check that regardless of method, the wide data is recovered via the `all.equal()` function, which checks for precisely what it states. To view the results, type `diris.wide` at the console:

```
diris.wide <- dcast(diris.long, ID + Species ~ Type,
                    value.var = list("Sepal", "Petal"), sep = ".")
all.equal(diris.wide2, diris.wide)
[1] TRUE
```

Summary

In this chapter, you met one of the two powerful (and more modern) ways of managing data in R. In particular, the type of data we manipulate with `data.table` is column and row, or table, data. In the next chapter, we delve more deeply into data manipulation. From there, you'll meet `dplyr`, which is the other modern way of managing data in R. We will close out this section with a look at databases outside R. Table 7-1 provides a summary of key functions used in this chapter.

Table 7-1. *Key Functions Described in This Chapter*

Function	What It Does
`d[i, j, by]`	General structure of a call to subset, compute in, or modify a data table.
`as.data.table()`	Coerces a data structure to a data table.
`setkey()`	Takes a data table in the first formal and then creates a key based on the column name arguments next passed to the function.
`haskey()`	This function is called on a data table and returns a Boolean value.
`key()`	Returns the data table's key if it exists.
`order()`	Controls the order to be ascending or descending for the data of a column.
`tables()`	Returns any current data tables currently active.
`anyDuplicated()`	Returns the location of the first duplicated entry.
`duplicated()`	Returns `TRUE` or `FALSE` for all rows to show duplication. Can take a key. Otherwise, it is by all columns.
`unique()`	Returns a data table that has only unique rows. Again, these may be specified to be specific rows. If there is a key, it is based on the key. If there is no key, it is based on all columns.
`setnames()`	Allows column names to be changed.
`setcolorder()`	Allows column order to be adjusted.
`:=`	Assignment operator within a data table.
`.()`	Shorthand for `list()`.
`.N`	Shorthand for number of rows in a data table, or number of rows in a subset, such as when a compute operation is performed by a grouping variable.
`merge()`	Merges two data tables based on matching key columns.
`melt()`	Converts wide data to long data.
`dcast()`	Converts long data to wide data.

CHAPTER 8

Data Munging with data.table

We already introduced the `data.table` package (Dowle, Srinivasan, Short, and Lianoglou, 2015). The `data.table` package is the heart of this chapter, covering the basics of accessing, editing, and manipulating data under the broad term *data management*. Although not glamorous, data management is a critical first step to data visualization or analysis. Furthermore, the majority of time on a particular analysis project often comes from data management. For example, running a linear model in R takes one line of code, once the data is clean and in the expected format. Data management is challenging because raw data comes in all types, shapes, and formats, and missing data is common. In addition, you may also have to combine or merge separate data sources. In this chapter, we go beyond the basic use of `data.table` to more-complex data management tasks.

There tend to be two stages to this sort of data wrangling. One-time conversions are often manual, as writing code often is not efficient if it is not reused (for example, changing one or two variable names, or renaming a data file to be consistent). For operations needing repetition (for example, renaming or labeling hundreds of variables) or working with larger data, more programming is used for data management. Add the `stringdist` package (van der Loo, 2014) to your `checkpoint` (Microsoft, 2016); another needed library is `foreign` (R Core Team, 2015). We run `checkpoint` as well as code to make data tables print in a neat fashion:

```
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")

library(stringsdist)
library(data.table)
library(foreign)
options(width = 70) # only 70 characters per line

options(stringsAsFactors = FALSE,
        datatable.print.nrows = 20,
        datatable.print.topn = 3,
        digits = 2)
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_8

Data Munging / Cleaning

To obtain data that is of the second stage of munging, we download data from the National Survey of Children's Health, 2003 (ICPSR 4691) at www.icpsr.umich.edu/icpsrweb/ICPSR/studies/4691, where we chose the Stata data format. We place the file in our working directory and use `read.dta()` to input the data to R. After converting it to a data table, we set the key to be the identification number column `IDNUMR`:

```
d <- read.dta("ICPSR_04691/DS0001/04691-0001-Data.dta")
Warning message:
In `levels<-`(`*tmp*`, value = if (nl == nL) as.character(labels) else paste0(labels,  :
  duplicated levels in factors are deprecated
d <- as.data.table(d)
setkey(d, IDNUMR)
```

To get a sense of the structure of this much data (there are over 300 variables), we use the `str()` function. We suppress attributes in order to focus on the column names and the types of data inside each column. Setting `strict.width = "cut"` enforces the options we set earlier, and we look at only the first `20` columns.

```
str(d, give.attr = FALSE, strict.width = "cut", list.len = 20)
Classes 'data.table' and 'data.frame': 102353 obs. of  301 variables:
 $ IDNUMR  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ STATE   : Factor w/ 51 levels "1-AK","2-AL",..: 35 25 45 18 36 37..
 $ MSA_STAT: Factor w/ 4 levels "-2 - MISSING",..: 4 4 4 4 4 4 4 4 4..
 $ AGEYR_CH: int  12 1 10 7 9 11 9 15 17 1 ...
 $ TOTKIDS4: Factor w/ 4 levels "1 - 1 CHILD",..: 1 1 1 3 4 2 3 3 1 ..
 $ AGEPOS4 : Factor w/ 5 levels "1 - ONLY CHILD",..: 1 1 1 4 2 2 3 2..
 $ S1Q01   : Factor w/ 8 levels "-4 - PARTIAL INTERVIEW",..: 6 5 6 5..
 $ RELATION: Factor w/ 6 levels "-2 - MISSING",..: 3 2 3 3 2 2 2 4 3..
 $ TOTADULT: Factor w/ 6 levels "-2 - MISSING",..: 4 3 2 3 2 3 3 2 4..
 $ EDUCATIO: Factor w/ 6 levels "-2 - MISSING",..: 4 4 4 4 3 4 4 3 4..
 $ PLANGUAG: Factor w/ 5 levels "-2 - MISSING",..: 2 2 2 2 2 2 2 2 2..
 $ S2Q01   : Factor w/ 11 levels "-4 - PARTIAL INTERVIEW",..: 5 5 5 ..
 $ S2Q02R  : int  59 29 49 45 96 64 51 64 65 33 ...
 $ HGHT_FLG: int  0 0 0 0 0 0 0 0 0 0 ...
 $ S2Q03R  : int  100 20 55 60 98 115 64 135 115 26 ...
 $ WGHT_FLG: int  0 0 0 0 0 0 0 0 0 0 ...
 $ BMICLASS: Factor w/ 5 levels "-2 - MISSING",..: 3 1 3 5 1 3 3 3 3..
 $ S2Q04   : Factor w/ 8 levels "-4 - PARTIAL INTERVIEW",..: 5 5 5 5..
 $ S2Q05   : Factor w/ 8 levels "-4 - PARTIAL INTERVIEW",..: 4 4 4 4..
 $ S2Q06   : Factor w/ 8 levels "-4 - PARTIAL INTERVIEW",..: 4 4 4 4..
  [list output truncated]
```

Many columns contain missing data that is currently stored as levels of a factor, rather than as R's `NA` to indicate to R that the values are missing. With over 100,000 rows, we definitely need to automate any changes. From the study's documentation, we can find the values used that we will recode to missing. These values have consistent labels, but their numbers change depending on the number of legitimate levels in a variable. We use the `table()` function to generate a frequency table of the unique values in a variable. This is useful both as a way to see the unique values and to become familiar with the data and how much is missing (relatively little for variables such as education, and more for BMI class). We want to recode partial interview, not in universe, missing, legitimate skip, don't know, and refused to `NA` in R. However, the labels are not identical—`EDUCATIO` has the number `97` for refused, whereas `S1Q01` has `7` for refused.

```
table(d[, EDUCATIO])
```

```
                    -2 - MISSING                     1 - LESS THAN HIGH SCHOOL
                               3                                          4661
2 - 12 YEARS, HIGH SCHOOL GRADUATE                     3 - MORE THAN HIGH SCHOOL
                           21238                                         76022
                96 - DON'T KNOW                                 97 - REFUSED
                             324                                           105
```

```
table(d[, PLANGUAG])
```

```
         -2 - MISSING            1 - ENGLISH 2 - ANY OTHER LANGUAGE         3 - DON'T KNOW
                    1                  94380                    7912                     51
          4 - REFUSED
                    9
```

```
table(d[, BMICLASS])
```

```
          -2 - MISSING          1-UNDERWEIGHT        2-NORMAL WEIGHT 3-AT RISK OF OVERWEIGHT
                 19963                   5921                  45650                   12207
          4-OVERWEIGHT
                 18612
```

```
table(d[, S1Q01])
```

```
-4 - PARTIAL INTERVIEW   -3 - NOT IN UNIVERSE           -2 - MISSING   -1 - LEGITIMATE SKIP
                     0                      0                      1                     0
             1 - MALE              2 - FEMALE         6 - DON'T KNOW           7 - REFUSED
                52554                   49719                     14                    65
```

Recoding Data

In addition to programmatically finding cases with values to recode to missing (for example, `7 - REFUSED` in `S1Q01`, and `97 - REFUSED` in `EDUCATIO`), we also want to drop those levels from the factor after converting them to missing. We can solve this by using regular expressions to search for the character strings we know. For example, we know that `REFUSED` always ends in that regardless of whether it starts with `7 -` or `97 -` or anything else. We can search using regular expressions via the `grep()` function, which returns the value or the numeric position of matches, or the `grepl()` function, which returns a logical (hence the `l`) vector of whether each element matched or not.

■ **Note** Regular expressions are a powerful tool for finding and matching character string patterns. In R, `grep()` and `grepl()` return the positions or a logical vector of which vector elements match the expected pattern. Combined with replacement, regular expressions help recode data.

We show these two functions in action with a simple example. The first formal, `pattern`, is the regular expression used for matching, while `x` is the data to search through. Notice the difference between logical vector versus numeric position. Either way, the second and fourth locations of `x` contain our matching letters of `abc`:

```
grep( pattern = "abc",  x = c("a", "abc", "def", "abc", "d"))
[1] 2 4
grepl( pattern = "abc", x = c("a", "abc", "def", "abc", "d"))
[1] FALSE  TRUE FALSE  TRUE FALSE
```

So far, we have a basic regular expression. We could use this for our data, searching for `REFUSED` instead of `abc`, although it is good to be as accurate as possible. For example, what if one variable has `1 - REFUSED TREATMENT`, `2 - DID NOT REFUSED TREATMENT`, `3 - REFUSED` (ignoring the grammar), where `1` and `2` are valid responses and `3` indicates a refusal to answer the question. A search for `REFUSED` would match all of those. We ultimately want to use `grepl()`, as logical values are useful indexes to set specific cases to `NA`. To ensure that we are grabbing the right values while checking that our regular expression is accurate, though, we use `grep()` with `value = TRUE`, which returns the matching strings. This makes it easy to see what we are matching.

```
grep( pattern = "REFUSED",
      x = c( "1 - REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT", "3 - REFUSED"),
      value = TRUE)
[1] "1 - REFUSED TREATMENT"    "2 - DID NOT REFUSED TREATMENT"        "3 - REFUSED"
```

To make it more specific, we could add the hyphen(`-`), which, although more precise, still gives us a false positive:

```
grep( pattern = "- REFUSED",
       x = c( "1 - REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT", "3 - REFUSED"),
       value = TRUE)
[1] "1 - REFUSED TREATMENT" "3 - REFUSED"
```

To go further, we specify that `- REFUSED` must be the last part of the string. That is, nothing can come after `- REFUSED`. This is done by using the `$` in regular expressions to signify the end of the pattern string:

```
grep( pattern = "- REFUSED$",
      x = c( "1 - REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT", "3 - REFUSED"),
      value = TRUE)
[1] "3 - REFUSED"
```

Now, your data may be different, so what we need is a variety of ways to refine just what type of matching we need to have. Regular expressions allow us to specify what we expect to find. The following pattern, although short, contains much information. The `[0-9]` signifies the digits `0` to `9`, and the `+` means the previous expression should occur one or more times (but not zero times). This code allows for both 1 and 97 but not a blank, and not anything but a number. After one or more numbers, the regular expression indicates that we expect to find `- REFUSED` and then the end of the string, which we specify using the dollar sign. We extend our possible values to match for our x value and specify our pattern precisely, with the results shown here:

```
grep( pattern = "[0-9]+ - REFUSED$",
      x = c( "1 - TREATMENT REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT",
            "3 - JACK - REFUSED", "4 - REFUSED", "97 - REFUSED", "- REFUSED"),
      value = TRUE)
[1] "4 - REFUSED"  "97 - REFUSED"
```

As a final step, we know that if we want REFUSED, the string starts with a number as well. Because it may be a negative number, we expand our previous expression to search for a negative operator zero or more times at the start of the string, followed by a number one or more times. We indicate the start of the string by using ^, and we indicate zero or more times by using *. By using this regular expression, we help protect against matching legitimate values in the data and errantly converting them to NA. We are quite specific in what we want, and anything that does not match our pattern is rejected.

```
grep( pattern = "^[-]*[0-9]+ - REFUSED$",
      x = c( "1 - TREATMENT REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT",
             "3 - JACK - REFUSED", "4 - REFUSED", "97 - REFUSED", "-97 - REFUSED",
             "- REFUSED", "TRICK CASE 4 - REFUSED"),
      value = TRUE)
[1] "4 - REFUSED"   "97 - REFUSED"  "-97 - REFUSED"
```

By default, regular expressions are also case sensitive. In-depth coverage of using regular expressions is beyond the scope of this book, but we explain some examples throughout this chapter. We recommend *Mastering Regular Expressions* by Jeffrey Friedl (O'Reilly Media, 2006) for readers who are interested in comprehensive coverage of regular expressions. Regular expressions are quite useful for matching and working with string data because they allow us to encode very specific searches.

Going back to our data, we want to match more than just REFUSED. One option would be to create individual regular expressions for each and loop through, but this is cumbersome and inefficient. Instead, we can modify our regular expression to indicate different options in some positions. This is done by using a pipe, |, which functions as *or*, to separate options. In the following code, we keep the specifics about the *start* of the string but allow REFUSED or MISSING to be the *ending* text:

```
grep( pattern = "^[-]*[0-9]+ - REFUSED$|MISSING$",
      x = c( "1 - TREATMENT REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT",
             "3 - JACK - REFUSED", "4 - REFUSED", "97 - REFUSED", "-97 - REFUSED",
             "-2 - MISSING", "- REFUSED", "TRICK CASE 4 - REFUSED", "4 - REFUSED HELP"),
      value = TRUE)
[1] "4 - REFUSED"   "97 - REFUSED"  "-97 - REFUSED" "-2 - MISSING"
```

Now we are ready to replace responses with NA for the patterns we want. As you see, the regular expression has become something rather complex. A word of caution is that many symbols have special meanings in regular expressions. Searching for special symbols requires special care. For instance, you have seen that * is used to indicate zero or more times, not a literal asterisk. To search for a literal character, it must be escaped by using a backslash (\). We now build our pattern that ought to identify all missing values we would like to see in our data set on child health. We pull our pattern out as a separate piece for readability and run one last test on our made-up x data. It is important to note that although the regular expression has a line break in order to fit in the book, it should be written as a single line with no space or line break before the NOT IN UNIVERSE$ portion.

```
p <- "^[-]*[0-9]+ - REFUSED$|MISSING$|DON'T KNOW$|LEGITIMATE SKIP$|PARTIAL INTERVIEW$|
      NOT IN UNIVERSE$"

grep( pattern = p,
     x = c( "1 - TREATMENT REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT",
            "3 - JACK - REFUSED", "4 - REFUSED", "97 - REFUSED", "-97 - REFUSED",
            "-2 - MISSING", "96 - DON'T KNOW", "-4 - LEGITIMATE SKIP", "-3 - PARTIAL
            INTERVIEW",
            "-2 - NOT IN UNIVERSE", "- REFUSED", "TRICK CASE 4 - REFUSED",
            "4 - PARTIAL INTERVIEW OF DOCTOR"),
     value = TRUE)
[1] "4 - REFUSED"          "97 - REFUSED"         "-97 - REFUSED"        "-2 - MISSING"
[5] "96 - DON'T KNOW"      "-4 - LEGITIMATE SKIP"  "-3 - PARTIAL INTERVIEW" "-2 - NOT IN UNIVERSE"
```

Now that we can find cases we want to set to missing, let's do it! Rather than type each variable, we loop through them in `data.table`, setting matching cases to `NA`. If it is a factor, we use `droplevels()` to remove unused factor levels and otherwise return them as is. To replace each variable, we use the `:=` operator introduced in the previous chapter in a new way. Previously, you saw how to create a variable in a data table as `dat[, NewVar := value]`. Now we are going to replace multiple variables at once. To do this, we pass a vector of variable names on the left of the assignment operator, `:=`, and a list of the values on the right-hand side. Because the variables already exist in the data set, we are overwriting them rather than creating new ones. The variable names are stored in the vector, `v`. We wrap `v` in parentheses so that `data.table` evaluates `v` as an R object name containing a vector of variable names. If we did not wrap `v` in parentheses, `data.table` would try to assign the results to a variable named `v`.

The next challenge is selecting the variables. We normally type unquoted variable names in a data table, but our variable names are stored in the vector `v`. To accomplish this, we leverage an internal object in `data.table`, `.SD`, which is essentially a list version of the data table. However, we do not want to loop through every single variable in the data table. Although `.SD` defaults to being all variables in the data table, we can make `.SD` contain only a subset of the variables in the data table by using the `.SDcols` argument. `.SDcols` is a formal argument that takes a character vector and uses that to control the variables included in the internal `.SD` object in a data table. Thus, we specify the variables to include in `.SD` by writing `.SDcols = v`. This operation does not return any easy-to-read values, so instead we return to `EDUCATIO` and `S1Q01`:

```
v <- c("EDUCATIO", "PLANGUAG", "BMICLASS", "S1Q01", "S2Q01")
d[, (v) := lapply(.SD, function(x) {
   x[grepl(pattern = p, x)] <- NA
  if (is.factor(x)) droplevels(x) else x }), .SDcols = v]

table(d[, EDUCATIO])

         1 - LESS THAN HIGH SCHOOL 2 - 12 YEARS, HIGH SCHOOL GRADUATE
                              4661                                21238
         3 - MORE THAN HIGH SCHOOL
                             76022
table(d[, S1Q01])

  1 - MALE 2 - FEMALE
     52554      49719
```

Note that there is some inefficiency because the data is copied each time for the function run in data table. It would be more efficient to select the relevant rows in `data.table` and then set those as missing. The difficulty here is that our results are factors, and we want to drop the excess levels or coerce them to characters. We did not coerce them to factors in the first place; the data came that way from the `Stata` data file. This is simply part of dealing with the data that we get. If we had all character data, we might have used a slightly more efficient idiom. If we wished, we might also have converted all factors to the character class at the beginning.

■ **Note** The `gsub()` function combines matching with regular expressions, and replaces the portions of a string that match the regular expression with new text in this form:

```
gsub(pattern = "regex", replacement = "new values", x = "original data").
```

Another recoding task we may want to do is to drop the numbers so that only the labels remain. This can be accomplished using the `gsub()` function, which performs regular expression matching within a string and then replaces matches with specified values (which can be a zero-length string). We first look at the following simple example:

```
gsub( pattern = "abc", replacement = "",  x = c("a", "abcd", "123abc456"))
[1] "a"      "d"      "123456"
```

Because it just uses regular expressions to match and remove numbers, we can reuse our pattern from before to test this out:

```
p.remove <- "^[-]*[0-9]+ - "
gsub( pattern = p.remove, replacement = "",
      x = c( "1 - TREATMENT REFUSED TREATMENT", "2 - DID NOT REFUSED TREATMENT",
             "3 - JACK - REFUSED", "4 - REFUSED", "97 - REFUSED", "-97 - REFUSED",
             "-2 - MISSING", "96 - DON'T KNOW", "-4 - LEGITIMATE SKIP",
             "-3 - PARTIAL INTERVIEW", "-2 - NOT IN UNIVERSE", "- REFUSED",
             "TRICK CASE 4 - REFUSED", "4 - PARTIAL INTERVIEW OF DOCTOR"))
 [1] "TREATMENT REFUSED TREATMENT" "DID NOT REFUSED TREATMENT"   "JACK - REFUSED"
 [4] "REFUSED"                     "REFUSED"                     "REFUSED"
 [7] "MISSING"                     "DON'T KNOW"                  "LEGITIMATE SKIP"
[10] "PARTIAL INTERVIEW"           "NOT IN UNIVERSE"             "- REFUSED"
[13] "TRICK CASE 4 - REFUSED"      "PARTIAL INTERVIEW OF DOCTOR"
```

For efficiency, we can combine this with our previous code to set some cases to missing. Doing it in one step avoids repetitive processing. First, though, we read the data back in so we have the raw data:

```
d <- read.dta("ICPSR_04691/DS0001/04691-0001-Data.dta")
d <- as.data.table(d)
setkey(d, IDNUMR)
```

Because `gsub()` coerces the input to characters regardless of whether we pass factors or characters to it, we will get characters out. So to know whether the input was a factor or not, we create an object, `f`, which is a logical indicating whether `x` started off as a factor. Then, rather than drop levels, we just convert them to factor if appropriate. Other than that change, we meld our earlier code with our removal code:

```
d[, (v) := lapply(.SD, function(x) {
   f <- is.factor(x)
   x[grepl(pattern = p, x)] <- NA
   x <- gsub(pattern = p.remove, replacement = "", x)
   if (f) factor(x) else x }), .SDcols = v]

table(d[, EDUCATIO])

12 YEARS, HIGH SCHOOL GRADUATE          LESS THAN HIGH SCHOOL          MORE THAN HIGH SCHOOL
                         21238                           4661                          76022
table(d[, S1Q01])

FEMALE   MALE
 49719  52554

table(d[, S2Q01])

EXCELLENT      FAIR      GOOD      POOR VERY GOOD
    65252      2189     10680       297     23903
```

Recoding Numeric Values

Because they have a more systematic structure, recoding *numeric* values is easier than recoding string data. To indicate different types of "missingness" in numeric data, people often use out-of-bounds values, such as a negative number or very high numbers (for example, 999). Coding different types of missing values makes sense from a data perspective, as the codes provide additional information (for example, skipped a question, not an applicable question). However, from a practical use perspective, we typically want to ignore all missing data (for example, when calculating the mean, it makes no sense to include -2 as a child's age when the parent refused to report it or -3 if a parent forgot). We can find values that fall between a range by using the `%between%` operator in the `data.table` package, and we can locate the complement by using `!` as before.

For example, for height, the documentation indicates that values zero or below, and above 90, are used to code various types of *missing*. Similarly, for weight, values zero or below, as well as above 900, are used to code different types of missing values. To tell R that these values indicate missing, we need to recode them. We see from our `table()` data that follows that there are indeed some missing values, and we can see exactly what values are used to code them (`-2`, `96`, `96`, `996`, `997`):

```
table(d[!S2Q02R %between% c(0, 90), S2Q02R])

  -2   96   97
   4 8096  131
table(d[!S2Q03R %between% c(0, 900), S2Q03R])

  -2  996  997
   4 2375   80
```

Again we could operate on these variables individually. However, that becomes cumbersome for many variables. The pattern often is that if valid values are less than 9, then 9 is missing. If values go into double digits, then >90 is missing, and so on. We recode programmatically by examining the maximum value and

setting the bounds accordingly. We see what the variable was like originally with the following code and in Figure 8-1. To select a single variable while maintaining a data table, rather than a numeric vector, we wrap the variable name as a list by using the `.()` shorthand for the `list()` function. The result is that the `hist()` function dispatches to methods for a data table, resulting in somewhat more elegant histograms:

```
hist(d[, .(S2Q03R)])
hist(d[, .(S2Q02R)])
```

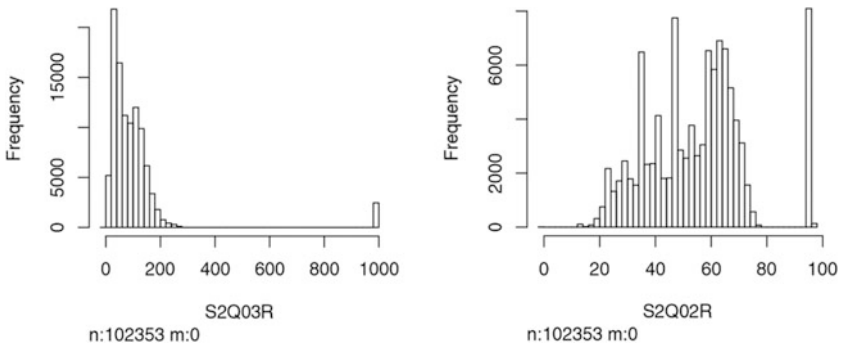


Figure 8-1. *Outliers that are near >900 and >90*

We focus on three columns that need numeric recoding. For each variable, we loop through calling the column name in our list. Designating `m` as our maximum, we start off with `j` and `i` set equal to `1`. As long as `j` is equal to `1` and `i` is less than or equal to the number of values to try (for example, 9, 99, 999), we keep doing the next calculation. If the maximum (ignoring missing) of the variable, `k`, is less than the maximum of the `i`th `m` value, then set `j` to `0` (which breaks the loop) and replace any values outside the range of `0` and the `i`th `m`. If the `i`th `m` is greater than `90` or minus some minuscule number, set the variable to missing (`NA_integer_`). In essence, we are counting through our list by columns, and we check whether the maximum value in our entire column is one of our maximum `m` values. Once it is there, we make sure that, effectively, values from 9 to 10, or 90 to 100, or 900 to 1000, are set to missing. Note that we use `NA_integer_` rather than `NA` because we are replacing a subset of values in an existing variable. `data.table` expects that the class of data being used to replace values within an existing variable match the class of that variable. We could use `NA` earlier because we overwrote the entire variable rather than replacing select values from within the variable.

```
v2 <- c("S2Q02R", "S2Q03R", "AGEYR_CH")
m <- sort(c(9, 99, 999))

for (k in v2) {
   j <- i <- 1
   while(j == 1 & i <= length(m)) {
     if (max(d[[k]], na.rm = TRUE) < m[i]) {
      j <- 0
      d[!(get(k) %between% c(0, ifelse(m[i] > 90,
        m[i] - 9, m[i] - 1e-9))), (k) := NA_integer_]
    } else { i <- i + 1   }
  }
}
```

We show the result of this after recoding by using the same histogram function as before and in Figure 8-2. Notice the `n` has been lowered a fair bit in both cases, and naturally the x-axis is much tighter.

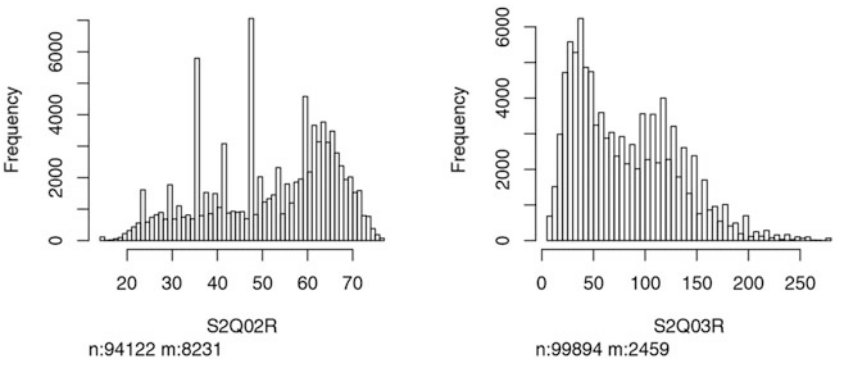


Figure 8-2. *The outliers are gone because histograms ignore missing values*

With this, we conclude our section about data hygiene. Next, we show how to create new variables to simplify the analysis process. In a data science perspective, variable creation is sometimes called *feature creation*. In many domains of social and behavioral research, questionnaires or surveys are commonly employed. In education, tests are administered. Regardless of whether the data is from tests, questionnaires, or surveys, a common task is to aggregate responses to individual items or variables to create a scale score, the overall test score, or some other aggregate index.

Creating New Variables

A questionnaire asks whether a doctor or health professional ever told the respondent that the focal child had any of nine possible conditions (for example, asthma, ADD or ADHD, depression or anxiety problems, diabetes, developmental delay, or physical impairment). Composite variable creation is the goal, to capture a number of issues. Some problems apply only to older children, and some respondents may not know the answer or may have refused to answer. Thus, we might take the average number of yes responses to calculate the proportion of yes responses out of all valid responses per respondent.

We first create a list of variables that pulls these nine columns' worth of data. From there, to get our actual variables, we `unlist()` them from our data table to see all possible responses that may require cleanup to reduce to a `Yes`/`No` scenario. Notice in the following code that there are several cases of missing data, legitimate skips, don't know, and refused:

```
v.health <- paste0("S2Q", c(19, 20, 21, 22, 23, 24, 26, 35, 37))
v.health
[1] "S2Q19" "S2Q20" "S2Q21" "S2Q22" "S2Q23" "S2Q24" "S2Q26" "S2Q35" "S2Q37"
table(unlist(d[, v.health, with = FALSE]))

-4 - PARTIAL INTERVIEW   -3 - NOT IN UNIVERSE           -2 - MISSING
                     0                       0                    120
  -1 - LEGITIMATE SKIP                 0 - NO                1 - YES
                 48388                  833454                  37720
       6 - DON'T KNOW             7 - REFUSED
                  1362                     133
```

We already know how to clean these sorts of variables by using regular expressions and `grep()`. After cleanup, we check all the responses again, including `NA`s in our table with the argument `useNA = "ifany"`.

It is important to note that although the regular expression has a line break to fit in this book, it should be written as a single line with no space or line break before the `NOT IN UNIVERSE$` portion.

```
p <- "^[-]*[0-9]+ - REFUSED$|MISSING$|DON'T KNOW$|LEGITIMATE SKIP$|PARTIAL INTERVIEW$|
      NOT IN UNIVERSE$"

d[, (v.health) := lapply(.SD, function(x) {
   x[grepl(pattern = p, x)] <- NA
   if (is.factor(x)) droplevels(x) else x
}), .SDcols = v.health]

table(unlist(d[, v.health, with = FALSE]), useNA = "ifany")
```

```
 0 - NO 1 - YES     <NA>
 833454   37720   50003
```

Now we can create a new variable that is the average of yes responses. To calculate the average by row, we use the `rowMeans()` function. We leverage the fact that logical values are stored as `FALSE = 0` and `TRUE = 1`. Thus, we test whether the character data is equal to `1 - YES`, which will return `TRUE` or `FALSE`, and then take the row means of those logical values. We ignore (drop) missing values by setting `na.rm = TRUE`.

```
d[, HealthConditions := rowMeans(d[, v.health, with = FALSE] == "1 - YES", na.rm = TRUE)]

table(round(d$HealthConditions, 2), useNA = 'ifany')
```

```
    0  0.11  0.12  0.14  0.17   0.2  0.22  0.25  0.29  0.33  0.38   0.4  0.43  0.44
77426 16116   239    20     5   676  4386   156    11  1933    75    81    12   768
  0.5  0.56  0.57   0.6  0.62  0.67  0.71  0.75  0.78  0.88  0.89   NaN
   51   264     3     3    13    80     1     5    19     1     3     6
```

We could follow a similar process to get the row counts. Looking at the counts, something has happened, though. There are no missing values! This is because with `rowSums()`, when `na.rm = TRUE` and a row has no valid data, its sum is zero, not a missing value. To correct this, we need to manually set to missing any cases/rows where all are missing:

```
d[, NHealthConditions := rowSums(d[, v.health, with = FALSE] == "1 - YES", na.rm = TRUE)]
table(d$NHealthConditions, useNA = 'ifany')
```

```
    0     1     2     3     4     5     6     7     8
77432 17068  4630  2018   820   278    84    20     3
```

We count the number of missing responses per respondent, select rows where the number missing equals the total, and set those to missing. Again note that when replacing a subset of values from a variable in `data.table`, match the class of missing to the class of the variable (for example, `NA_integer_` for integer, `NA_real_` for numeric, `NA_character_` for character, `NA` for logical). That gives us a better count:

```
d[, NMissHealthConditions := rowSums(is.na(d[, v.health, with = FALSE]))]
d[NMissHealthConditions == length(v.health), NHealthConditions := NA_integer_]
table(d$NHealthConditions, useNA = 'ifany')
```

```
    0     1     2     3     4     5     6     7     8  <NA>
77426 17068  4630  2018   820   278    84    20     3     6
```

Using `rowMeans()` or `rowSums()` to create new variables is a relatively elegant solution. However, it is not a very `data.table` approach. The calculation occurs outside the original `data.table`; we call the object, `d`, a second time and subset the columns for use. When calling the data object a second time, we lose access to many features of `data.table`, such as performing operations for only certain subsets of the data, or performing an operation by some grouping factor.

If ignoring missing values is not an issue, we can easily work directly with `data.table` by using the `Reduce()` function. `Reduce()` takes a function (typically, a binary operator) as its first argument, and a vector or list of arguments. We show examples here of a vector and of a list with addition, division, and powers:

```
Reduce(`+`, c(1, 2, 3))
[1] 6
Reduce(`+`, list(1:3, 4:6, 7:9))
[1] 12 15 18
Reduce(`/`, list(1:3, 4:6, 7:9))
[1] 0.036 0.050 0.056
Reduce(`^`, list(1:3, 4:6, 3:1))
[1]    1 1024  729
```

This can then easily be applied to variables in a `data.table`. We cannot directly add the health variables, because they are factor class data. However, we could write a function to deal with it:

```
fplus <- function(e1, e2) {
   if (is.factor(e1)) {
     e1 <- as.numeric(e1) - 1 }
   if (is.factor(e2)) {
     e2 <- as.numeric(e2) - 1 }
   e1 + e2
}
```

We select the columns by using `.SDcols`, and then `.SD` will be a list of variables that we can reduce and store results in a new variable:

```
d[, NHealthConditions2 := Reduce(fplus, .SD), .SDcols = v.health]
table(d$NHealthConditions2)

    0     1     2     3     4     5     6     7     8
65459 16116  4386  1927   768   264    79    19     3
```

Fuzzy Matching

Approximate string matching, or *fuzzy matching*, is a technique whereby observations link to a reference list. For example, you may have a list of registered users, and then individuals write their names on an attendance sheet. Alternately, you may be working with filenames that are supposed to match certain pieces of information.

We start with two lists; the reference names are our registered user list. These are the names we believe in and want to match to the observed names that were written in quickly on our hypothetical attendance sheets. Users may have attended more than one event and thus show up multiple times, or may not show up at all. Also note that some of these words have double spaces between them, which are required to match our later output.

```
reference.Names <- c("This Test", "Test Thas",
  "Jane Mary", "Jack Dun-Dee")
observed.Names <- c("this test", "test this", "test that",
  "JaNe  Mary.", "Mary Sou", "Jack Dee", "Jane Jack")
```

The challenge is to find out the number of events that registered users attended (that is, signed). A first pass can be done by using the `stringdistmatrix()` function from the `stringdist` package. We use the Damerau-Levenshtein distance method, which essentially counts the number of characters that have to change to go from one string to another. From this, we can see for each observed name the minimum number of characters that must be changed to match one of our reference names. For `this test`, it takes two character changes to match the first reference name, eight for the second reference name, and so on. Note that by default, `stringdistmatrix()` is case sensitive, so switching cases counts as one change. Notice that this matrix gives us a grid of those matches, where the matrix elements are the distance:

```
stringdistmatrix(reference.Names, observed.Names, method = "dl")
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    2    8    7   10    8    7    8
[2,]    8    3    3    9    8    8    8
[3,]    8    8    8    3    7    6    3
[4,]   11   11   11    9   10    4    9
```

If we want only matches, we can use approximate matching. It takes similar arguments, but also the maximum distance allowed before calling nothing a match. Notice that it returns positions in the reference name vector:

```
amatch(observed.Names, reference.Names, method = "dl", maxDist = 4)
[1]  1  2  2  3 NA  4  3
```

We can expand a bit and make it non-case-sensitive by converting to lowercase via `tolower()`, or to uppercase via `toupper()`. Additionally, we can remove some things we do not care about, such as various punctuation marks. Notice that both of these methods have a chance to give us lower distances overall (although in our case, there are no periods to remove):

```
stringdistmatrix( tolower(reference.Names), tolower(observed.Names), method = "dl")
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    6    5   10    8    7    8
[2,]    6    1    1    9    8    8    8
[3,]    8    8    8    2    7    6    3
[4,]   11   11   11    9   10    4    9

stringdistmatrix( tolower(gsub("\\.", "", reference.Names)),
                  tolower(gsub("\\.", "", observed.Names)), method = "dl")
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]    0    6    5    9    8    7    8
[2,]    6    1    1    8    8    8    8
[3,]    8    8    8    1    7    6    3
[4,]   11   11   11    9   10    4    9
```

We can split a character string by a particular character by using `strsplit()`. Any pattern or character can be used, but we use spaces. We use *backslash s* (`\s`) as a special character in regular expressions to mean any type of space. A second backslash is required in order to escape the first backslash. That is because the special character is not `s`, which would be escaped as `\s`, but rather the special character is the compound `\s`, thus the escaped version is `\\s`. We demonstrate the result of `strsplit()` in the following code:

```
strsplit( x = reference.Names[1], split = "\\s")[[1]]
[1] "This" "Test"

strsplit( x = observed.Names[2], split = "\\s")[[1]]
[1] "test" "this"
```

Now we can use `stringdistmatrix()` again. The result shows that each subchunk of the second observed name is one character manipulation away from one of the subchunks from the first reference name:

```
stringdistmatrix(
   strsplit( x = reference.Names[1], split = "\\s")[[1]],
   strsplit( x = observed.Names[2],  split = "\\s")[[1]],
   method = "dl")
     [,1] [,2]
[1,]    4    1
[2,]    1    4
```

If we ignore case, we get perfect matches. The second chunk of observed name 2 matches the first chunk of reference name 1, and the first chunk of observed name 2 perfectly matches the second chunk of the reference name 1:

```
stringdistmatrix(
   strsplit( x = tolower(reference.Names[1]), split = "\\s")[[1]],
   strsplit( x = tolower(observed.Names[2]),  split = "\\s")[[1]],
   method = "dl")
     [,1] [,2]
[1,]    3    0
[2,]    0    3
```

To return the sum of the best matches, we can pick the minimum and sum. The results show us that ignoring case and ignoring ordering makes the second observed name a perfect match for the first reference name. We would never have gotten this result if we tested it as an overall string.

```
sum(apply(stringdistmatrix(
   strsplit( x = tolower(reference.Names[1]), split = "\\s")[[1]],
   strsplit( x = tolower(observed.Names[2]),  split = "\\s")[[1]],
   method = "dl"), 1, min))
[1] 0
```

Combining everything you have learned, yields a list of techniques to use for string matching. We find values to ignore (for example, punctuation), split strings (for example, on spaces), and ignore case. We write a function to help with matching names. To this, we also add an optional argument, `fuzz`, to control whether to include close matches within a certain degree of tolerance. We also count exact matches. This code is not

optimized for speed or efficiency, but is an example combining many aspects of what you've learned. It also applies that knowledge to a set of names, where each written name (index argument) compares against a vector of possible candidates (the pool). We comment the function inline to explain the pieces, rather than write a wall of text at the beginning. We also do not bold this code because of its length.

```
matchIt <- function(index, pool, ignore = c("\\.", "-"), split = FALSE,
                    ignore.case = TRUE, fuzz = .05, method = "dl") {

## for those things we want to ignore, drop them, e.g., remove spaces, periods, dashes
  rawpool <- pool

  for (v in ignore) {
    index <- gsub(v, "", index)
    pool <- gsub(v, "", pool)
  }

## if ignore case, convert to lowercase
  if (ignore.case) {
    index <- tolower(index)
    pool <- tolower(pool)  }

  if (!identical(split, FALSE)) {
    index2 <- strsplit(index, split = split)[[1]]
    index2 <- index2[nzchar(index2)]
    pool2 <- strsplit(pool, split = split)
    pool2 <- lapply(pool2, function(x) x[nzchar(x)])

## calculate distances defaults to the Damerau-Levenshtein distance
    distances <- sapply(pool2, function(x) {
      sum(apply(stringdistmatrix(index2, x, method = method),
                1, min, na.rm = TRUE))
    })
  } else {
    ## calculate distances defaults to the Damerau-Levenshtein distance
    distances <- stringdist(index, pool, method = method)
  }

## some methods result in Infinity answers, set these missing
  distances[!is.finite(distances)] <- NA

## get best and worst
  best <- min(distances, na.rm = TRUE)
  worst <- max(distances, na.rm = TRUE)

## if fuzz, grab all distances within fuzz percent of the difference between best and worst
  if (fuzz) {
    usedex <- which(distances <= (best + ((worst - best) * fuzz)))
  } else {
    usedex <- which(distances == best)  }
```

```
## define a distance below which it is considered a perfect or exact match
  perfect <- distances[usedex] < .01
  out <- rawpool[usedex]

## count the number of perfect matches
  count <- sum(perfect)

##the function continues onto the next page##
  if (any(perfect)) {
  ## if there are perfect matches, use just one
    match <- out[perfect][1]
## return a data table of the perfect match and number of perfect matches (probably 1)
    data.table(
      Match = match,
      N = count,
      Written = NA_character_)
  } else {
## if no perfect match, return list of close matches, comma separated and exactly as written
    data.table(
      Match = paste(out, collapse = ", "),
      N = count,
      Written = index)
  }
}
```

Now we loop through each observed name and try to match it to reference names within 5 percent of the best match (`.05`). Since the output is always the same, we can combine it row-wise.

```
output <- lapply(observed.Names, function(n) {
  matchIt( index = n, pool = reference.Names, ignore = c("\\.", "-"), split = "\\s", fuzz = .05)
})

output <- do.call(rbind, output)
output
                     Match N   Written
1:               This Test 1        NA
2:               This Test 1        NA
3:               Test Thas 0 test that
4:               Jane Mary 1        NA
5:               Jane Mary 0  mary sou
6:            Jack Dun-Dee 0  jack dee
7: Jane Mary, Jack Dun-Dee 0 jane jack
```

We can see that `This Test` shows up in two rows, so we aggregate up and print the final output. There are two perfect matches for `This Test`. There is no perfect match for `Test Thas`, although there is a close one listed under the `Uncertain` column. Finally, `jack dee` is a close but uncertain match for `Jack Dun-Dee`, and `jane jack` is an uncertain match for `Jane Mary` or `Jack Dun-Dee`.

```
finaloutput <- output[, .(
  N = sum(N),
  Uncertain = paste(na.omit(Written), collapse = ", ")),
  by = Match]
finaloutput
                    Match N Uncertain
1:               This Test 2
2:               Test Thas 0 test that
3:               Jane Mary 1  mary sou
4:            Jack Dun-Dee 0  jack dee
5: Jane Mary, Jack Dun-Dee 0 jane jack
```

Although it is challenging to cover every example and use case, these basic tools should be enough to cover many types of data management tasks when used in combination.

Summary

In this chapter, you saw how to locate text based on either exact regular expressions or fuzzy matching. Additionally, you learned how to substitute new string pieces for old. Finally, we discussed ways of creating new variables in `data.table`. Table 8-1 summarizes the key functions presented in this chapter.

Table 8-1. *Key Functions Described in This Chapter*

Function	What It Does
`read.dta()`	Reads Stata files
`str()`	Displays R object structure
`grep()`	Pattern matching that returns a vector showing which elements match
`grepl()`	Pattern matching that returns a vector showing Boolean for all values in x
`gsub()`	Matches a pattern and makes a substitution by returning the original vector after making changes
`rowMeans(), rowSums()`	Calculates the average or sum of values for each row of a data table or matrix
`Reduce()`	Takes a binary function as its first argument and applies that to its remaining argument
`.SD`	Reserved name within `data.table` that can be used to refer to all the variables within the data table as a list
`.SDcols`	Argument to `data.table` that allows specification of the variables to be included in `.SD`
`strsplit()`	Splits a string based on a particular splitting character
`stringdistmatrix()`	Creates a matrix that has distances between observations and expected values
`amatch()`	Provides a vector that shows which index in the expected values most closely matches the observed values
`tolower()`	Coerces all characters in the given string to lowercase

CHAPTER 9

Other Tools for Data Management

Comparing data frames and data tables leads to an interesting question. What if there were more types of data? Particularly, what if there were different ways to store data that are all, at their heart, tables of some sort? In addition to data frames or tables, there are many ways to store data, many of which are just tables. The idea behind `dplyr` (Wickham and Francois, 2015) is that regardless of what the data back end might be, our experience should be the same. To allow this, `dplyr` implements generic functions for common data management tasks. For each of these generic functions, specific methods are written that translate the generic operation into whatever code or language is required for a specific back end. Using a layer of abstraction ensures that users get a consistent experience, regardless of the specific data format, or back end, being used. It also makes `dplyr` extensible, in that support for a new format can be added by simply writing additional functions or methods. The user experience need not change. For this chapter, our `checkpoint` header needs to have both the `tibble` and the `dplyr` packages installed and added:

```
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
library(dplyr)
library(tibble)
```

The `dplyr` package names functions after verbs that accomplish a specific action. That is, rather than have a few functions that each perform many tasks, `dplyr` has many functions that typically accomplish only a specific task. Unlike `data.table` (Dowle, Srinivasan, Short, and Lianoglou, 2015), `dplyr` is *functional* in that it always takes input and returns output, in this case, data. It does not modify the data in place in memory as `data.table` does. However, it is still quite fast, because many of the functions are optimized, written in C++, and `dplyr` tries not to make unnecessary copies.

As before, we set some options to reduce the number of rows printed. The default data format for `dplyr` is a `tibble` (Wickham, Francois, and Müller, 2016). Tibbles are implemented in the `tibble` package. Essentially *tibbles* are an extension of data frames, which have some additional restrictions (such as defaulting to character strings rather than coercing character strings to factors) and nicer methods for printing or displaying on the screen. Data manipulation is handled by `dplyr`, while `tibble` handles the actual storage of data. In technical terms, the `tibble` package implements the R object `class` as well as some basic methods for tibbles.

```
options(stringsAsFactors = FALSE,
        tibble.print_max = 20, ## if over 20 rows
        tibble.print_min = 5, ## print first 5
        digits = 2) ## reduce digits printed by R
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_9

```
iris$Species <- as.character(iris$Species)
diris <- as_tibble(iris)
diris
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
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
# ... with 145 more rows
```

For `data.table`, we use `sapply()` to get a vector of classes for each column or variable. Although we could do the same with tibbles, tibbles automatically provide information about the types or classes of data in each column as well as total number of columns and rows (for example, `<dbl>` indicating a double or numeric value, with `<chr>` indicating character class data). We turn our attention to the common goals of working with data: sorting, subsetting, ordering, computing, and reshaping. A helpful way to think of `dplyr` is that it uses verbs to describe actions on the data.

Sorting

We sort in `dplyr` by the `arrange()` function, which takes a data set as its first argument. Additional arguments after the first argument provide sorting by those other columns. The default is an increasing order, and this is modified with a second function call of `desc()` to provide **desc**ending order.

```
diris <- arrange(diris, Sepal.Length)
diris
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         2.9          1.4         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         3.2          1.3         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
# ... with 145 more rows
diris <- arrange(diris, Sepal.Length, desc(Sepal.Width))
diris
# A tibble: 150 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
# ... with 145 more rows
```

It is possible to see the first-row index when a duplicate occurs for either the entire data set or for just a particular column, via `anyDuplicated()`. When calling this function on the entire data set, it looks for a match between all columns, which occurs for rows 78 and 79 in this data set. A duplicated variable in sepal length occurs in the third row, as shown in the following code:

```
anyDuplicated(diris)
[1] 79

anyDuplicated(diris$Sepal.Length)
[1] 3
```

Whereas `anyDuplicated()` returns the row index of the first duplication, `duplicated` simply tells us how many such duplications occur in our table. Looking at our code, we see that for a full match in all columns, row 79 is the only time that occurs. On the other hand, duplicated sepal lengths may happen for the first time in row 3, but there are many more duplicates.

```
table(duplicated(diris))
FALSE  TRUE
  149     1
table(duplicated(diris$Sepal.Length))
FALSE  TRUE
   35   115
```

If we want to see table results with only distinct values, `distinct()` works much like `arrange()`. The first formal is for data, while the rest specify column name variables. If no column names are specified, `distinct()` operates on all columns. Recall from the `duplicated()` call that there are 149 unique rows.

```
distinct(diris)
# A tibble: 149 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
# ... with 144 more rows
```

Similarly, recall that there are only 35 unique rows if we consider only sepal length. Consequently, using the second formal of `distinct()` to hold the `Sepal.Length` column yields only 35 rows in our `tibble`. We show the first five of those rows here:

```
distinct(diris, Sepal.Length)
# A tibble: 35 x 1
  Sepal.Length
         <dbl>
1          4.3
2          4.4
3          4.5
4          4.6
5          4.7
# ... with 30 more rows
```

This section closes with one last call to `distinct()`, showing more than one column called in the second and third formal arguments. Here, a row is unique provided both `Sepal.Length` and `Sepal.Width` do not duplicate. Pay special attention to rows 2 through 4, which have the same `Sepal.Length`, yet those rows are not duplicated because their `Sepal.Width`s are different:

```
distinct(diris, Sepal.Length, Sepal.Width)
# A tibble: 117 x 2
  Sepal.Length Sepal.Width
         <dbl>       <dbl>
1          4.3         3.0
2          4.4         3.2
3          4.4         3.0
4          4.4         2.9
5          4.5         2.3
# ... with 112 more rows
```

Selecting and Subsetting

Selecting portions of the stored data in `dplyr` is different from the `data.table` methodology. Remember, `dplyr` works via function calls. These functions are designed to be independent of the underlying data structure. Nevertheless, we do want the same end results. Selecting rows by position is accomplished using `slice()`, which takes two formal arguments. The first is the data object, in our case `diris`, and the second allows us to select specific rows. We first select just the first five rows:

```
slice(diris, 1:5)
# A tibble: 5 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
```

As seen before, `-` can be used for negation. Therefore, we could also drop all rows but the last five rows by dropping rows 1 to 145:

```
slice(diris, -(1:145))
# A tibble: 5 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width   Species
         <dbl>       <dbl>        <dbl>       <dbl>     <chr>
1          7.7         3.8          6.7         2.2 virginica
2          7.7         3.0          6.1         2.3 virginica
3          7.7         2.8          6.7         2.0 virginica
4          7.7         2.6          6.9         2.3 virginica
5          7.9         3.8          6.4         2.0 virginica
```

Rather than select rows directly, we may set logical conditions on them. Again, these types of arguments belong in the second formal of our function call. However, in this case, we use `filter()` rather than `slice()`. The `filter()` function uses logical indexing. It is easy enough to select either all the rows in which `Species` either is (or is not) `setosa` by writing a logical test:

```
filter(diris, Species == "setosa")
# A tibble: 50 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
# ... with 45 more rows
 filter(diris, Species != "setosa")
# A tibble: 100 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
         <dbl>       <dbl>        <dbl>       <dbl>      <chr>
1          4.9         2.5          4.5         1.7  virginica
2          4.9         2.4          3.3         1.0 versicolor
3          5.0         2.3          3.3         1.0 versicolor
4          5.0         2.0          3.5         1.0 versicolor
5          5.1         2.5          3.0         1.1 versicolor
# ... with 95 more rows
```

It is also possible to select rows based on numeric values with inequalities, equality, or nonequality in addition to logical indexes. Both numeric and character tests may be mixed and matched to extract precisely the rows desired:

```
filter(iris, Sepal.Length < 5 & Petal.Width > .2)
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1          4.6         3.4          1.4         0.3     setosa
2          4.5         2.3          1.3         0.3     setosa
3          4.8         3.0          1.4         0.3     setosa
4          4.9         2.4          3.3         1.0 versicolor
5          4.9         2.5          4.5         1.7  virginica
filter(diris, Sepal.Length == 4.3 | Sepal.Width != 4.4)
# A tibble: 149 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
5          4.5         2.3          1.3         0.3  setosa
# ... with 144 more rows
```

If we seek multiple matches, the %in% operator may be used along with a list of interest to us. This allows us to separate the coding that indicates our choice of interest, and then filter based on that interest:

```
interest <- c(4.3, 4.4)
filter(diris, Sepal.Length %in% interest)
# A tibble: 4 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3         3.0          1.1         0.1  setosa
2          4.4         3.2          1.3         0.2  setosa
3          4.4         3.0          1.3         0.2  setosa
4          4.4         2.9          1.4         0.2  setosa
```

Alternately, if we wish to exclude any rows that have a certain characteristic, that may be done via negation. This is useful to remove outliers or erroneous data points:

```
filter(diris, !Sepal.Length %in% interest)
# A tibble: 146 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.5         2.3          1.3         0.3  setosa
2          4.6         3.6          1.0         0.2  setosa
3          4.6         3.4          1.4         0.3  setosa
4          4.6         3.2          1.4         0.2  setosa
5          4.6         3.1          1.5         0.2  setosa
# ... with 141 more rows
```

The last filter example we show uses several arguments. Using multiple arguments is not required, but can be a helpful way to break code into human-readable chunks. Multiple arguments are logically equivalent to &.

```
filter(diris,
       Sepal.Length == 4.3 | Sepal.Length == 4.4,
       Petal.Width < .2)
# A tibble: 1 x 5
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
         <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1          4.3           3          1.1         0.1  setosa
```

So far, we've selected rows from all columns. If we wish to select only specific columns, the select() function takes a database as its first formal, and column information in the rest of the formals. For named columns in your table data structure, selecting one or more columns is as simple as listing the column names. It is also possible to select by position. Contrast these named versus position column arguments and also recognize that both can make the column selections in any order needed:

```
select(diris, Sepal.Length, Sepal.Width)
# A tibble: 150 x 2
  Sepal.Length Sepal.Width
         <dbl>       <dbl>
1          4.3         3.0
2          4.4         3.2
```

```
3          4.4         3.0
4          4.4         2.9
5          4.5         2.3
# ... with 145 more rows
select(diris, 1, 5, 2)
# A tibble: 150 x 3
  Sepal.Length Species Sepal.Width
         <dbl>   <chr>       <dbl>
1          4.3  setosa         3.0
2          4.4  setosa         3.2
3          4.4  setosa         3.0
4          4.4  setosa         2.9
5          4.5  setosa         2.3
# ... with 145 more rows
```

So far, specific columns are directly named. What if the columns need to be variable depending on the program or the user's needs? In that case, character string references are desirable, as strings easily convert to variables. We show both a direct call to `select_()` and the use of a variable `v` to achieve the same results. Be sure to notice the underscore added to the name of the function! When using `select_()`, multiple variable names as strings can be passed as separate arguments.

```
select_(diris, "Sepal.Length", "Sepal.Width")
# A tibble: 150 × 2
  Sepal.Length Sepal.Width
         <dbl>       <dbl>
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
# ... with 145 more rows
```

Our variable, `v`, could contain more than one column name. However, if used in the default way, only the first variable from the list will be returned. To get all the variables to be returned, when passing an object holding a vector of variable names, it should be passed to the `.dots` formal argument, shown in the second example that follows:

```
v <- c("Sepal.Length", "Sepal.Width")
select_(diris, v)
# A tibble: 150 x 1
  Sepal.Length
         <dbl>
1          4.3
2          4.4
3          4.4
4          4.4
5          4.5
# ... with 145 more rows
```

```
select_(diris, .dots = v)
# A tibble: 150 x 2
  Sepal.Length Sepal.Width
         <dbl>       <dbl>
1          5.1         3.5
2          4.9         3.0
3          4.7         3.2
4          4.6         3.1
5          5.0         3.6
# ... with 145 more rows
```

Another way of achieving the same result with our original `select()` function is by using one of the select helper functions. There are seven of these helper functions, and the one we presently want is `one_of()`, which manages character vectors:

```
select(diris, one_of(v))
# A tibble: 150 x 1
  Sepal.Length
         <dbl>
1          4.3
2          4.4
3          4.4
4          4.4
5          4.5
# ... with 145 more rows
```

When using the `dplyr` data structure `tibble`, approaches like those used in data tables work. These are not guaranteed to translate as smoothly throughout the `dplyr` experience. It seems best to engage `dplyr` in its native language. However, the next two function calls are given for completeness:

```
diris[, v]
# A tibble: 150 x 1
  Sepal.Length
         <dbl>
1          4.3
2          4.4
3          4.4
4          4.4
5          4.5
# ... with 145 more rows
diris[, 1]
# A tibble: 150 x 1
  Sepal.Length
         <dbl>
1          4.3
2          4.4
3          4.4
4          4.4
5          4.5
# ... with 145 more rows
```

So far, all our work returns another `tibble` as output. Extracting the raw data as a vector uses several techniques. If we know the column name containing the data we want, it is easy to type the name directly. If our column may be variable, we reuse our variable `v` for the third method. Regardless of the method, the console prints the same data. Notice that these data are vectors, not data table structures. We marked the code to access the column data as a vector in bold only for emphasis.

```
head(diris[["Sepal.Length"]])
[1] 4.3 4.4 4.4 4.4 4.5 4.6
head(diris$Sepal.Length)       # easy to type
[1] 4.3 4.4 4.4 4.4 4.5 4.6
head(diris[[v]])               # easy if you have a R variable
[1] 4.3 4.4 4.4 4.4 4.5 4.6
```

Excluding a column or columns may be of more interest than selecting certain columns. Exclusion uses negation in several formats. Particularly, either the column(s) to be excluded may be named or referenced via position:

```
select(diris, -1)
# A tibble: 150 x 4
  Sepal.Width Petal.Length Petal.Width Species
        <dbl>        <dbl>       <dbl>   <chr>
1         3.0          1.1         0.1  setosa
2         3.2          1.3         0.2  setosa
3         3.0          1.3         0.2  setosa
4         2.9          1.4         0.2  setosa
5         2.3          1.3         0.3  setosa
# ... with 145 more rows
select(diris, -Sepal.Length, -Petal.Width)
# A tibble: 150 x 3
  Sepal.Width Petal.Length Species
        <dbl>        <dbl>   <chr>
1         3.0          1.1  setosa
2         3.2          1.3  setosa
3         3.0          1.3  setosa
4         2.9          1.4  setosa
5         2.3          1.3  setosa
# ... with 145 more rows
```

Again, when variable exclusion is needed, we set a variable to hold the vector, and we may paste negation to the front of that by using `paste0()` and then using the `select_()` call to use character strings. Alternately, we may use negation with the select helper function `one_of()` in our usual `select()` function. Either way, we get the same results:

```
ex <- paste0("-", v)
select_(diris, ex)
# A tibble: 150 x 4
  Sepal.Width Petal.Length Petal.Width Species
        <dbl>        <dbl>       <dbl>   <chr>
1         3.0          1.1         0.1  setosa
2         3.2          1.3         0.2  setosa
3         3.0          1.3         0.2  setosa
```

```
4          2.9           1.4          0.2  setosa
5          2.3           1.3          0.3  setosa
# ... with 145 more rows
select(diris, -one_of(v))
# A tibble: 150 x 4
  Sepal.Width Petal.Length Petal.Width Species
        <dbl>        <dbl>       <dbl>   <chr>
1         3.0          1.1         0.1  setosa
2         3.2          1.3         0.2  setosa
3         3.0          1.3         0.2  setosa
4         2.9          1.4         0.2  setosa
5         2.3          1.3         0.3  setosa
# ... with 145 more rows
```

Having now used this select convenience function a couple of times, we look briefly at four more of these. The goal of these four is that they help us choose variables without typing out an entire column name. In no particular order, `starts_with(), ends_with(), contains(), and matches()` all use regular expressions, default to looking in the current variable list, and ignore letter case by default, although this is a feature that may be turned off. Rather than demonstrate all of these, the following code shows two examples using `starts_with()`. One benefit of the `dplyr` approach of making focused functions named after verbs that accomplish one specific task is that it is easy to tell what a function does from its name. The remaining helper functions operate similarly to `starts_with()` and do exactly what their names indicate:

```
select(diris, starts_with("s"))
# A tibble: 150 x 3
  Sepal.Length Sepal.Width Species
         <dbl>       <dbl>   <chr>
1          4.3         3.0  setosa
2          4.4         3.2  setosa
3          4.4         3.0  setosa
4          4.4         2.9  setosa
5          4.5         2.3  setosa
# ... with 145 more rows
select(diris, starts_with("s", ignore.case = FALSE))
# A tibble: 150 x 0
```

With this last bit of code, our journey through `dplyr` subsetting ends. We turn our attention to variable renaming and ordering.

Variable Renaming and Ordering

It is a regrettably routine part of data hygiene that column names require renaming or reordering. What made sense (or is "legal") in one database may not work in another data bank. Many proprietary data management systems have fixed names that are unwieldy.

Viewing the names associated with a dplyr object requires the usual names() or colnames() functions called on our data set diris. Either way, we get the same results with the current column order:

```
names(diris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
colnames(diris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

To change a name, call the rename() function on the data object and set the new name in the second formal by using the following format: new name = old name. Multiple columns can be renamed at once by passing them as additional arguments (for example, in the third, fourth formal). Confirm that the change took place by comparing the results of name() with the old results. In the following code, we remove the full stop between Sepal and Length in our data:

```
diris <- rename(diris, SepalLength = Sepal.Length)
names(diris)
[1] "SepalLength"  "Sepal.Width"  "Petal.Length" "Petal.Width"  "Species"
```

To reorder the columns to pair the lengths and the widths together, simply use the select() function and reassign a name to your data object. Be sure to remember that we renamed SepalLength to no longer have a full stop in the middle!

```
diris <- select(diris, SepalLength, Petal.Length, Sepal.Width, Petal.Width, Species)
diris
# A tibble: 150 x 5
  SepalLength Petal.Length Sepal.Width Petal.Width Species
        <dbl>        <dbl>       <dbl>       <dbl>   <chr>
1         4.3          1.1         3.0         0.1  setosa
2         4.4          1.3         3.2         0.2  setosa
3         4.4          1.3         3.0         0.2  setosa
4         4.4          1.4         2.9         0.2  setosa
5         4.5          1.3         2.3         0.3  setosa
# ... with 145 more rows
```

Earlier, you saw that numeric select calls could reorder columns, so we close out this section with one last reordering example. Here, we use a variable to hold numeric positions, although we could just as well use character names in place of numbers. Then we use the select_() function to reorganize our columns back to their original order.

```
v <- c(1, 3, 2, 4, 5)
diris <- select_(diris, .dots = as.list(v))
diris
# A tibble: 150 x 5
  SepalLength Sepal.Width Petal.Length Petal.Width Species
        <dbl>       <dbl>        <dbl>       <dbl>   <chr>
1         4.3         3.0          1.1         0.1  setosa
2         4.4         3.2          1.3         0.2  setosa
3         4.4         3.0          1.3         0.2  setosa
4         4.4         2.9          1.4         0.2  setosa
5         4.5         2.3          1.3         0.3  setosa
# ... with 145 more rows
```

Computing on Data and Creating Variables

Now that you have a good sense of how to manage original data in `dplyr`, we'll start to focus on how to create new data from old data. For simplicity's sake, we re-create our `diris` data set from the original `iris` data set. Then we'll be ready to use the new function `mutate()` to create or replace columns. This function takes the usual first argument, and the latter formals express what is done to create new columns. Whether it is one or many, simply create expressions that let `dplyr` know what to append to the `tibble`:

```
diris <- mutate(diris, V0 = 0, X1 = 1L, X2 = 2L)
diris
# A tibble: 150 x 8
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species    V0    X1    X2
         <dbl>       <dbl>        <dbl>       <dbl>   <chr> <dbl> <int> <int>
1          5.1         3.5          1.4         0.2  setosa     0     1     2
2          4.9         3.0          1.4         0.2  setosa     0     1     2
3          4.7         3.2          1.3         0.2  setosa     0     1     2
4          4.6         3.1          1.5         0.2  setosa     0     1     2
5          5.0         3.6          1.4         0.2  setosa     0     1     2
# ... with 145 more rows
```

Of course, new data might not be about creating wholly new columns. Instead, we may create new calculated columns based on existing information. Perhaps we want a new column named `V` that is the multiplication of `Petal.Length` and `Petal.Width`. For clarity's sake, we first remove the three columns just added previously:

```
diris <- select(diris, -V0, -X1, -X2)
diris <- mutate(diris, V = Petal.Length * Petal.Width)
diris
# A tibble: 150 x 6
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species     V
         <dbl>       <dbl>        <dbl>       <dbl>   <chr> <dbl>
1          5.1         3.5          1.4         0.2  setosa  0.28
2          4.9         3.0          1.4         0.2  setosa  0.28
3          4.7         3.2          1.3         0.2  setosa  0.26
4          4.6         3.1          1.5         0.2  setosa  0.30
5          5.0         3.6          1.4         0.2  setosa  0.28
# ... with 145 more rows
```

More exotic calculations are performed by doing them only when certain conditions occur. In this case, `if_else()` takes three arguments. The first is the logical condition on which to test the if/else statement. The second and third formals indicate what to do on the `if` or the `else`. Additionally, `mutate()` allows us not only to create a column with data, but also to change existing column data. Thus, if it was not enough in our first pass to create a column `V2` that took on a multiplicative relationship between petal widths and lengths for `setosa` species, we could extend that to a square-root operation for `virginica`. To make sure we see a sample of each species, we use the `slice()` function to look at just three specific rows:

```
diris <- mutate(diris, V2 = if_else(Species == "setosa", Petal.Length * Petal.Width, NA_
real_))
diris <- mutate(diris, V2 = if_else(Species == "virginica",
                                    sqrt(Petal.Length * Petal.Width), V2))
slice(diris, c(1, 51, 101))
```

```
# A tibble: 3 x 7
  Sepal.Length Sepal.Width Petal.Length Petal.Width    Species     V    V2
         <dbl>       <dbl>        <dbl>       <dbl>      <chr> <dbl> <dbl>
1          5.1         3.5          1.4         0.2     setosa  0.28  0.28
2          7.0         3.2          4.7         1.4 versicolor  6.58    NA
3          6.3         3.3          6.0         2.5  virginica 15.00  3.87
```

It is not necessary to create a new column; it is possible to simply calculate information on a table by using the `summarise()` function. The first formal, as always, tells the function which data to use, while the rest are used to hold individual calculations. Note that one or many calculations might be coded. Here, we demonstrate two such calculations for arithmetic mean and standard deviation:

```
summarise(diris, M = mean(Sepal.Width), SD = sd(Sepal.Width))
# A tibble: 1 x 2
      M    SD
  <dbl> <dbl>
1   3.1  0.44
```

Unlike `data.table`, `dplyr` works via functions. Moreover, while your mathematics professor may have spent some time lecturing about function composition (and that technique of nesting functions inside each other has its place), there is a cleaner look in `dplyr`. Chaining together several functions leaves us with the desired results when using the `%>%` operator. Suppose we want to take our data, use only the `virginica` species, and calculate the mean of the sepal widths. The result should be a `tibble` of just one variable, the mean we seek. Note that when the `%>%` operator is used, the data set is passed to the first argument. Thus, even though it looks like we specify the first formal argument to `filter()` in the following code, because we used the `%>%` operator, the function is modified so that the data set is passed as the first argument, and what we wrote, `Species == "virginica"`, is passed as the second argument:

```
diris  %>%  filter(Species == "virginica") %>%  summarise(M = mean(Sepal.Width))
# A tibble: 1 x 1
      M
  <dbl>
1     3
```

This idea of taking it one step at a time can be modified to manage all sorts of data manipulations. Thinking back to our study of `data.table`, the third formal of that structure, `by =`, allowed us to organize by elements of a column. For `dplyr`, we use `group_by()` on a column name to accomplish the same task. Expanding our mean example to all three species, we now have a two-column `tibble` with one row for each species. We could also get means for more than one of our original columns. Notice that `virginica`'s sepal width mean is 3, regardless of how we calculate it.

```
diris %>%  group_by(Species) %>%  summarise(M1 = mean(Sepal.Width), M2 = mean(Petal.Width))
# A tibble: 3 x 3
     Species    M1    M2
       <chr> <dbl> <dbl>
1     setosa   3.4  0.25
2 versicolor   2.8  1.33
3  virginica   3.0  2.03
```

This chaining structure, just like the data table chaining structure, can be done indefinitely (or close to it). To find a correlation in the preceding data, we simply add one more step to the end of our chain:

```
diris %>%
  group_by(Species) %>%
  summarise( M1 = mean(Sepal.Width),  M2 = mean(Petal.Width)) %>%
  summarise(r = cor(M1, M2))
# A tibble: 1 x 1
      r
  <dbl>
1 -0.76
```

Variables can create other variables as well. We create a new variable based on whether a petal width is greater or smaller than the median petal width for that species. This may have changed its appearance a bit, but this is still a `tibble`:

```
diris <- diris %>%
  group_by(Species) %>%
  mutate(MedPW = Petal.Width > median(Petal.Width))
diris
Source: local data frame [150 x 8]
Groups: Species [3]

  Sepal.Length Sepal.Width Petal.Length Petal.Width Species     V    V2 MedPW
         <dbl>       <dbl>        <dbl>       <dbl>   <chr> <dbl> <dbl> <lgl>
1          5.1         3.5          1.4         0.2  setosa  0.28  0.28 FALSE
2          4.9         3.0          1.4         0.2  setosa  0.28  0.28 FALSE
3          4.7         3.2          1.3         0.2  setosa  0.26  0.26 FALSE
4          4.6         3.1          1.5         0.2  setosa  0.30  0.30 FALSE
5          5.0         3.6          1.4         0.2  setosa  0.28  0.28 FALSE
# ... with 145 more rows
```

It is also easy to group by multiple variables. Grouping by both species and median petal width Booleans, we recalculate means for sepal and petal widths:

```
diris  %>%
  group_by(Species, MedPW) %>%
  summarise( M1 = mean(Sepal.Width),  M2 = mean(Petal.Width))
Source: local data frame [6 x 4]
Groups: Species [?]

     Species MedPW    M1    M2
       <chr> <lgl> <dbl> <dbl>
1     setosa FALSE   3.4  0.19
2     setosa  TRUE   3.5  0.38
3 versicolor FALSE   2.6  1.19
4 versicolor  TRUE   3.0  1.50
5  virginica FALSE   2.8  1.81
6  virginica  TRUE   3.1  2.27
```

Again, it is always possible to chain on more operations:

```
diris %>%
  group_by(Species, MedPW) %>%
  summarise(
     M1 = mean(Sepal.Width),
     M2 = mean(Petal.Width)) %>%
  group_by(MedPW) %>%
  summarise(r = cor(M1, M2))
# A tibble: 2 x 2
  MedPW     r
  <lgl> <dbl>
1 FALSE -0.78
2  TRUE -0.76
```

Merging and Reshaping Data

Without repeating what we said about merging and reshaping data in the `data.table` chapter, we note that the function call to `merge()` still works, after a fashion. This function is the same function from `data.table`, and because we used `dplyr` to operate on `tibbles`, which are built on data frames, it works. However, when this is done, they lose their `tibble` status. Thus, we also introduce the `*join()` functions, which are much closer to some of the more traditional database languages.

Once again, we first run a bit of code to make four separate data objects. Please note again that this is not likely to be needed in real life, as our data usually comes from different databases:

```
diris <- diris  %>%  group_by(Species) %>%  select(Species, Sepal.Width) %>%  slice(1:3)
diris
Source: local data frame [9 x 2]
Groups: Species [3]

     Species Sepal.Width
       <chr>       <dbl>
1     setosa         3.5
2     setosa         3.0
3     setosa         3.2
4 versicolor         3.2
5 versicolor         3.2
6 versicolor         3.1
7  virginica         3.3
8  virginica         2.7
9  virginica         3.0

diris2 <- slice(diris, c(1, 4, 7))
diris2
Source: local data frame [3 x 2]
Groups: Species [3]
```

```
     Species Sepal.Width
       <chr>       <dbl>
1     setosa         3.5
2 versicolor         3.2
3  virginica         3.3
```

```
dalt1 <- tibble(
   Species = c("setosa", "setosa", "versicolor", "versicolor",Data managementmerging and
reshaping Data
               "virginica", "virginica", "other", "other"),
   Type = c("wide", "wide", "wide", "wide",
            "narrow", "narrow", "moderate", "moderate"),
   MedPW = c(TRUE, FALSE, TRUE, FALSE, TRUE, FALSE, TRUE, FALSE))
dalt1
# A tibble: 8 x 3
     Species     Type MedPW
       <chr>    <chr> <lgl>
1     setosa     wide  TRUE
2     setosa     wide FALSE
3 versicolor     wide  TRUE
4 versicolor     wide FALSE
5  virginica   narrow  TRUE
6  virginica   narrow FALSE
7      other moderate  TRUE
8      other moderate FALSE
```

```
dalt2 <- slice(dalt1, c(1, 3))
dalt2
# A tibble: 2 x 3
     Species  Type MedPW
       <chr> <chr> <lgl>
1     setosa  wide  TRUE
2 versicolor  wide  TRUE
```

The `merge()` function operates on two data sets. For one-to-one data such as `diris2` and `dalt2`, the merge by itself keeps only the rows that match both data sets. We can see that for these two objects, `virginica` row in `diris2` is dropped:

```
merge(diris2, dalt2)
     Species Sepal.Width Type MedPW
1     setosa         3.5 wide  TRUE
2 versicolor         3.2 wide  TRUE
```

If we wish to keep all three rows of `diris2`, then there is some missing data after the merge. Now, in this case, since we arbitrarily choose `diris2` to be our first formal, `all.x = TRUE` keeps all of the `diris2` rows:

```
merge(diris2, dalt2, all.x = TRUE)
     Species Sepal.Width Type MedPW
1     setosa         3.5 wide  TRUE
2 versicolor         3.2 wide  TRUE
3  virginica         3.3 <NA>    NA
```

It is also possible to set the second value, or `all.y = TRUE`, although in this case that has no noticeable difference. Depending on how good a match two particular tables are, there might be a different pattern of missing variables. Similarly, it is also possible to say that we want all rows from all tables. This has the potential to give the most missing data locations; in our example, such a merge would be identical to our three rows shown in the preceding example.

As mentioned, these are not `tibbles`. Now, we may wrap them in `as_tibble()` if we wish to merge and coerce back to a `tibble`. However, `dplyr` also operates on database-style objects, as you'll see in our database chapter. In particular, Structured Query Language (SQL) can be accessed through `dplyr`. SQL has joins, and we'll look at four types of joins now. To generalize greatly, `merge()` focuses on matching in some way. Joins focus on rows versus columns of object 1 and object 2.

All rows of the first object are returned in a `left_join()` as well as all columns of both the first and second objects. Because this is a `dplyr` function call, we get a `tibble` as our output. As you can see, this is similar (although not a perfect match in all cases) to `all.x=TRUE`:

```
left_join(diris2,dalt2)
Joining, by = "Species"
Source: local data frame [3 x 4]
Groups: Species [?]

     Species Sepal.Width  Type MedPW
       <chr>       <dbl> <chr> <lgl>
1     setosa         3.5  wide  TRUE
2 versicolor         3.2  wide  TRUE
3  virginica         3.3  <NA>    NA
```

A `right_join()` is much like a left, except it returns all rows from the second object. Still, all columns from both objects return. In this case, there is no major difference between this type of join and the `merge()` function. However, this case is not all cases. Many charts on the internet claim to speak the truth about how joins work. In our experience, the accuracy of those explanations can be hit or miss. Our advice is to grab several small data tables, or cook up your own as we did here. Then start joining things. Somewhere along the way, after doing it often, you'll gain a sense of which technique gets you closest to having the data you want.

```
right_join(diris2,dalt2)
Joining, by = "Species"
Source: local data frame [2 x 4]
Groups: Species [?]

     Species Sepal.Width  Type MedPW
       <chr>       <dbl> <chr> <lgl>
1     setosa         3.5  wide  TRUE
2 versicolor         3.2  wide  TRUE
```

The closest match to the basic `merge()` function is the `inner_join()` function, which in this case would yield the same results as the previous right join. Contrastingly, from a merge point of view, the weirdest join is the `anti-join()`. The `anti_join()` returns all rows in the first database that do not have matching values in the second data object. As for columns, it returns only the first object's columns.

```
anti_join(diris2,dalt2)
Joining, by = "Species"
Source: local data frame [1 x 2]
Groups: Species [3]
```

```
    Species Sepal.Width
      <chr>       <dbl>
1 virginica         3.3
```

Between joins and merges, there are plenty of options for combining data into a single object. Remember, your goal is to eventually get just the data needed for analysis. Sometimes it makes sense to first combine one or more collections of data into a single table, and from there use the techniques from earlier in this chapter to reduce that to your desired data set. Of course, if merges are more familiar, they often work just fine. If you truly require `dplyr`, `as_tibble()` always works as a wrapper function to coerce your object back to what it should be. After all, `data.table` and `dplyr`'s `tibbles` are both extensions of data frames.

Our last goal is reshaping, which works better if there is a unique ID per row. We start by refreshing our data set one more time. From there, we introduce `n()`, which is a special way to refer to the number of rows, much like `.N` is in data tables:

```
diris <- as_tibble(iris)
diris <- mutate(diris, ID = 1:n() )
diris
# A tibble: 150 x 6
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species    ID
         <dbl>       <dbl>        <dbl>       <dbl>   <chr> <int>
1          5.1         3.5          1.4         0.2  setosa     1
2          4.9         3.0          1.4         0.2  setosa     2
3          4.7         3.2          1.3         0.2  setosa     3
4          4.6         3.1          1.5         0.2  setosa     4
5          5.0         3.6          1.4         0.2  setosa     5
# ... with 145 more rows
```

To `reshape()` long, we use the base R function. From there, we call out repeatedly measured variables or variables we wish to stack in the long data set with the `varying` label. The label `timevar` takes the name of the variable indicating to which stack a row belongs. It is so named because often these are sorted based on time. Specify names for the newly stacked variables with `v.names` and use `idvar` for the ID variable. Any variables not listed in `varying` will be assumed to be not varying and will simply be repeated.

```
diris.long <- as_tibble(reshape(as.data.frame(diris),
  varying = list(
    c("Sepal.Length", "Sepal.Width"),
    c("Petal.Length", "Petal.Width")),
  timevar = "Type",
  v.names = c("Sepal", "Petal"),
  idvar = c("ID"),
  direction = "long"))

diris.long
# A tibble: 300 x 5
  Species    ID  Type Sepal Petal
*   <chr> <int> <int> <dbl> <dbl>
1  setosa     1     1   5.1   1.4
2  setosa     2     1   4.9   1.4
3  setosa     3     1   4.7   1.3
4  setosa     4     1   4.6   1.5
5  setosa     5     1   5.0   1.4
# ... with 295 more rows
```

At this point, it might not be clear what Type is. Type tells us whether we are in the length or width of sepals and petals. When our data stacks long from wide, it doubles in length. Now, the Sepal and Petal columns must do double duty, and Type is there to let us know which. Take a look at this slice() of our data:

```
slice(diris.long, c(1, 151))
# A tibble: 2 x 5
  Species    ID  Type Sepal Petal
    <chr> <int> <int> <dbl> <dbl>
1  setosa     1     1   5.1   1.4
2  setosa     1     2   3.5   0.2
```

Type varies from 1 to 2 depending on whether it is length or width. However, that information is not coded into our data, so we mutate() our tibble to make sure it understands Type is a factor of two levels, one of which is Length and the other Width. Notice that our slice now makes much more sense after this cleanup.

```
diris.long <- mutate(diris.long, Type = factor(Type, levels = 1:2,
                           labels = c("Length", "Width")))
diris.long
# A tibble: 300 x 5
  Species    ID   Type Sepal Petal
    <chr> <int> <fctr> <dbl> <dbl>
1  setosa     1 Length   5.1   1.4
2  setosa     2 Length   4.9   1.4
3  setosa     3 Length   4.7   1.3
4  setosa     4 Length   4.6   1.5
5  setosa     5 Length   5.0   1.4
# ... with 295 more rows

slice(diris.long, c(1, 151))
# A tibble: 2 x 5
  Species    ID   Type Sepal Petal
    <chr> <int> <fctr> <dbl> <dbl>
1  setosa     1 Length   5.1   1.4
2  setosa     1  Width   3.5   0.2
```

We took wide data and made it long. We now reverse the process by using reshape() again, this time with the direction formal set wide. Since we are going wide, we do not need varying, as we are about to get those through this process. We do need ids = diris.long$ID so that rows 1 and 151 are combined into the same single, wide row.

```
diris.wide2 <- as_tibble(reshape(as.data.frame(diris.long),
                                 v.names = c("Sepal", "Petal"),
                                 timevar = "Type",
                                 idvar = "ID",
                                 ids = diris.long$ID,
                                 direction = "wide"))
diris.wide2
# A tibble: 150 x 6
```

```
  Species    ID Sepal.Length Petal.Length Sepal.Width Petal.Width
*   <chr> <int>        <dbl>        <dbl>       <dbl>       <dbl>
1  setosa     1          5.1          1.4         3.5         0.2
2  setosa     2          4.9          1.4         3.0         0.2
3  setosa     3          4.7          1.3         3.2         0.2
4  setosa     4          4.6          1.5         3.1         0.2
5  setosa     5          5.0          1.4         3.6         0.2
# ... with 145 more rows
```

At this point, you can see that `reshape()` is part of base R and is not a very `dplyr` way of doing things. However, it is perhaps more powerful and versatile. Hence we decided to showcase how to use it rather than more `dplyr`-type approaches such as `tidyr`. The truth is, `tidyr` is around two years old, and it likely will become the new way of managing such conversions. Given the speed at which technology progresses, by the time you, our gentle reader, have this book in hand, all of us will be ready for an update to this chapter.

Summary

The `dplyr` and `tibble` packages make working with data easier. By separating what we do to data from the data itself, we can learn a comparatively limited number of manipulations that we can apply to any data. Behind the scenes, `dplyr` translates those function calls to modify the data. In the next chapter, you'll see how this allows us to quickly access big data. Table 9-1 describes the key functions used in this chapter.

Table 9-1. *Key Functions Described in This Chapter*

Function	What It Does
`as_tibble()`	Converts a data object (for example, a data frame) to a `tibble`.
`arrange()`	Arranges a named column in increasing order, or descending order if `desc()` is used on the column name.
`distinct()`	This is `dplyr`'s answer for `unique()`. It gives just those rows that are not the same.
`slice()`	Slices out the rows asked for from a tibble.
`filter()`	Filters out rows based on matched characteristics.
`select()`	Selects specific columns based on name. Functionality can be expanded by using the helper functions: `one_of()`, `starts_with()`, `ends_with()`, `contains()`, `matches()`, `num_range()`, and `everything()`.
`select_()`	A more programming-friendly version of `select()` that takes character strings or R objects containing character strings.
`rename()`	Renames columns.
`mutate()`	Allows for changes or mutations to existing tibbles. Adding/deleting/modifying column data is a popular use.
`summarise()`	Creates a new tibble based on calculations from an original tibble.
`%>%`	Piping avoids function composition and allows for cleaner code to chain up multiple functions.
`group_by()`	Allows grouping by columns. Corresponds to the third formal (`by =`) of `data.table`.
`left_join()`	Joins are a SQL way of dealing with data. A left-join keeps the first object's rows.

(*continued*)

Table 9-1. (*continued*)

Function	What It Does
`right_join()`	Keeps the second object's rows.
`inner_join()`	Returns all columns of both variables and all rows from the first that have matches in the second data object.
`anti_join()`	Returns just the unique, unlinkable part of the first data object. This is quite helpful if the second object is meant to extend information on the first, as it gives a fast way to see what is left.
`reshape()`	Depending on whether `direction = "wide"` or `direction = "long"` is chosen, reshapes data wide or long. This is a base R function and to keep a tibble requires `as_tibble()`.

CHAPTER 10

Reading Big Data(bases)

Now that you understand how to manage data inside R, let's consider where data is found. While smaller data is found in comma-separated values (CSV) files or files easily converted to such, larger data tends to live in other places. This chapter deals with big data, or at least data that may be big. What is the challenge with big data? R works in memory, random access memory, not hard drives. A quick check of your system settings should reveal the amount of memory you have. We, the authors, use between 4 and 32 gigabytes in our real-world systems, with the larger number being a somewhat expensive habit. R cannot analyze data larger than available RAM.

This leads to two possibilities. While it is possible to save R data objects and load them one at a time, R is not designed to be a full-time database. Furthermore, large data is often managed by other groups besides analysts. Big data is a reality, and most of it lives inside a database of some sort. Now, while most databases have the capability to export files to CSV, this is not always convenient. Our goal is to provide you with just enough information to allow you access to some popular and common databases. If you are a data user, you'll be able to access data from several common databases by the end of this chapter. If you are a big data owner, you'll gain the ability to write newly collected data into a database. Along the way, we'll show the commands allowing you to load into R a subset of your data. We also discuss read versus write privileges. Finally, (actually, first), we install these databases with you, so you are ready to practice using large data. Fair warning: Our goal is not to provide expertise in database management. We do have some book recommendations we make throughout the chapter for more details.

This chapter has three main sections that each follow the same pattern: installation of database software followed by interacting with that database via R. The first two database systems are SQLite and PostgreSQL, which are both relational databases. As such, they fit nicely with the previous chapters discussing data, because so far we've used tabular data. Tabular data is fixed columns and variable rows. New data is slotted neatly into a new row, and thus most data has a rigid format. Additionally, while there may be more than one table, there tend to be common keys that allow us to match tables to each other (and rows within those tables to each other). The last database we demonstrate is MongoDB, which is an example of a document store. In a document store, each piece, or *blob*, of data may be unique. There is no reason to believe that any keys or relations exist; of course, there is no reason to believe that there aren't any similarities either.

In this chapter, we use the following packages: `devEMF` (Johnson, 2015), `DBI` (R-SIG-DB, Wickham, and Kirill, 2016), `RSQLite` (Wickham, James, and Falcon 2014), `jsonlite` (Ooms, 2014), `tm` (Feinerer and Hornik, 2015), `wordcloud` (Fellows, 2014), `RMongo` (Cheng, 2013), `RPostgreSQL` (Conway, Eddelbuettel, Nishiyama, Prayaga, and Tiffin, 2016), and `data.table` (Dowle, Srinivasan, Short, and Lianoglou, 2015). The following code loads the `checkpoint` (Microsoft Corporation, 2016) package to control the exact version of R packages used and then loads the packages:

```
## load checkpoint and required packages
library(checkpoint)
checkpoint("2016-09-04", R.version = "3.3.1")
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_10

```
library(devEMF)
library(RPostgreSQL)
library(DBI)
library(RSQLite)
library(jsonlite)
library(tm)
library(wordcloud)
library(RMongo)
library(data.table)
options(width = 70) # only 70 characters per line
```

This chapter is not about curating big data and certainly is not about curating big data in R. Instead, this chapter gets you learning the ins and outs of using R with some popular open source databases. Before moving into specific databases, a word of advice: should you wish to learn all three of these databases, go to each section for installation instructions. Follow those, and, after you are finished, just to be safe, restart your computer.

SQLite

SQLite is both extraordinarily small and exceptionally efficient. It is also freely available (as of this writing and for the last 16 years). While we recommend reading *The Definitive Guide to SQLite* by Grant Allen and Mike Owens (Apress, 2010), there are five main data types recognized (compare this with R): null, integer, real, text, and blob.

Each SQLite database exists as just one file and is thus quite portable. These databases have no users, no need for passwords, and there is a simple connection structure. Disadvantages are a consequence of some of those same advantages. Without user privileges, there is no distinction between just reading information and being able to modify data.

This lack of user privileges deserves an important warning. We build the databases we use in this chapter with mock data. In real-life work, databases are not for practice. There is a healthy amount of fear when directly connecting to a live database for the first time, especially if you are one wrong drop call away from deleting your team's or client's data. Because SQLite is just one file, it is wise to try on a copy of the actual data first if at all possible. Alternatively, as in our case, practice first on something completely safe.

Installing SQLite on Windows

Installing does not happen with SQLite. Rather, the file(s) are simply downloaded. A visit to SQLite at `www.sqlite.org/download.html` is enough; we downloaded the `sqlite-tools-win32-VERSION.zip` file, where `VERSION` maps to the current product and version. Next, extract `sqliteV.exe`, where `V` is the version to a folder on the disk where you want your data to live. We extracted to `C:/sqlite` as our folder. Make sure you have free space on the disk; this database eventually takes up some space.

That is all it takes to get SQLite up and running on your system. Now, in real life, perhaps someone else created the SQLite database, and you have a database copy or access via a network share drive. Our goals are to use data from the database and to store data to the database from R. This is not the same, nor as efficient, as managing an SQLite database by using the "proper" techniques. All the same, it gets us using the database quickly!

SQLite and R

Our first task is connecting to our database. Connections use the `DBI` package library and build a link to our database. In the case of SQLite, there is no need for a username or password. We do need to tell the link the location of our database as well as the type of database. Recall that we made some choices about where to locate SQLite, and that location must match up with our current location. We call this first connection `con1` and then use the `dbIsValid()` function from the `RSQLite` library to test whether it started correctly. Ours returns as valid, so we are ready to go:

```
con1 = dbConnect(SQLite(), dbname = "C:/sqlite/LiteDB.db")
dbIsValid(con1)
[1] TRUE
```

Our goal is to get you connecting to this database quickly. We push our familiar iris data to SQLite, which allows us to see that the system is working correctly. Data in SQLite lives in tables, and inside a table are fields. This corresponds to a data table in R, which has a name and columns. We assume that you are already overly familiar with iris and show the result of writing the data table version of iris to SQLite. The `dbWriteTable()` function takes three required arguments: the connection, the name of the new table, and the name of the R object to write into that table. The R object should be a data frame. Additional arguments may be passed, including settings determining whether we overwrite or append, should a table already be in the database with the same name. For this first attempt, we set both to their defaults of `FALSE`:

```
dbWriteTable(con1, name = "Iris", diris, overwrite = FALSE, append = FALSE)
[1] TRUE
dbListTables(con1)
[1] "Iris"
dbListFields(con1, "Iris")
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width" "Species"
```

Now this data is in our SQLite `Iris` table; we can read it back into R. Our goal is for you to learn how to read an entire table into R and to confirm that our storage was lossless or accurate. Using the function `dbReadTable()` on our `con1` connection and our `Iris` table inside SQLite gives us access to our data. This function returns a data frame:

```
diris_lite1 <- dbReadTable(con1, "Iris")
diris_lite1 <- as.data.table(diris_lite1)
all.equal(diris, diris_lite1)
[1] TRUE
```

The `all.equal()` function confirms that there was no loss. This opens up an easy way to store large data. Data tables from R easily store, and through proper use of `overwrite`, `append`, or `name`, we have a good-enough way to curate that data with minimal direct knowledge of SQLite. Should you find yourself in possession of large tabular data, this is admittedly an easy way to keep track of it. One of the authors has a series of tabular data that changes over time on occasion but is highly similar. An extra column of the current date allows large chunks to be readily written to SQLite into a sort of permanent archive. Now, the main reason this might be useful is to read into R not an entire table, but merely part of one.

Reading part of a table is our goal, and it is not quite so easy as what we did previously. Depending on your familiarity with Structured Query Language (SQL), this step may be more or less easy. If you are less familiar with SQL, that is perfectly fine. Our advice, in that case, is first to cast a broad net into your SQLite table, and then use R to remove more rows or columns you neither need nor want. On the other hand, if you

are more comfortable with SQL, there is no reason not to select precisely just the parts you want. Here, we give a few examples that ought to be enough for beginning levels of SQL comfort. Again, we recommend other books for moving deeper into SQL.

A basic SQL query `SELECT`s specific columns, `FROM` a specific table, `WHERE` specific rows of specific columns have a certain feature. We use `dbSendQuery()` to send and execute our query to and in SQLite. Then, we use `dbFetch()` to get our results into R. In the end, we use `dbClearResult()` to clear the buffer in SQLite in preparation for any future queries.

```
query <- dbSendQuery(con1, "SELECT [Sepal.Length] FROM Iris WHERE Species = 'setosa'")
diris_lite2 <- dbFetch(query, n=-1)
head(diris_lite2)
  Sepal.Length
1          5.1
2          4.9
3          4.7
4          4.6
5          5.0
6          5.4
dbClearResult(query)
[1] TRUE
```

The preceding code returns `50` observations of sepal length of the `setosa` species. In `dbFetch()`, we reference the already executed query, and the `n` argument tells us the number of rows to select. In this case, we ask for all rows. Contrastingly, we might have asked for all `100` rows that do not have `setosa`. The command would be similar, as shown in the following code:

```
query <- dbSendQuery(con1, "SELECT [Sepal.Length], Species FROM Iris WHERE Species <>
'setosa'")
diris_lite3 <- dbFetch(query, n=-1)
head(diris_lite3)
Sepal.Length    Species
1          7.0 versicolor
2          6.4 versicolor
3          6.9 versicolor
4          5.5 versicolor
5          6.5 versicolor
6          5.7 versicolor
dbClearResult(query)
[1] TRUE
```

Notice that because of the full stop between `Sepal` and `Length`, we have to encase that variable in a bracket, unlike species. It is worth mentioning that SQL has some linguistic quirks and reserved characters (much as R does, to some degree). In fact, this could well be the biggest risk for those of us less experienced in SQL. Remember, as soon as you use `dbSendQuery`, SQLite cheerfully receives and executes that query. Since SQLite has no user authentication, there is no way to use minimal, read-only privileges. In the best case, the wrong commands simply have no effect. In the worst case,, you could delete, overwrite, or otherwise damage data. Our hard-earned advice is always to work on a local copy of your data, especially if you are reading this to start your big data project. Otherwise, talk to the database owner or administrator if you are new to SQL.

Our last iris example returns all columns by using the * wildcard rather than listing out all columns. Again, we ask for all rows that are not `setosa`:

```
query <- dbSendQuery(con1, "SELECT * FROM Iris WHERE Species <> 'setosa'")
diris_lite4 <- dbFetch(query, n=-1)
diris_lite4 <- as.data.table(diris_lite4)
diris_lite4
     Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
  1:          7.0         3.2          4.7         1.4 versicolor
  2:          6.4         3.2          4.5         1.5 versicolor
  3:          6.9         3.1          4.9         1.5 versicolor
 ---
 98:          6.5         3.0          5.2         2.0  virginica
 99:          6.2         3.4          5.4         2.3  virginica
100:          5.9         3.0          5.1         1.8  virginica
dbClearResult(query)
```

The purpose perhaps of SQLite is to allow for data larger than RAM might physically allow. It can be read in chunks as we've demonstrated, or we can push large amounts of data from R to free up memory. Our next example comes with a choice. If you have enough available free space in your system RAM, go ahead and do the example of 200 million rows. Otherwise, either change the number of variables in `rnorm()` to something lower, or use our code online to run the smaller example of 70 million rows.

This code creates 200 million entirely made-up data observations with a mean of 1 and a default standard deviation of 1. As you see, it is about 1.6 gigabytes in size. Certainly, data can be and is larger, but this is good enough as a test case for you to see the effects on your system. It likely takes about 5 minutes to run on your system, so be prepared for a bit of a wait.

```
BiggerData <- rnorm(200000000, 1)
BiggerData <-as.data.table(BiggerData)
object.size(BiggerData)
1600001088 bytes
dbWriteTable(con1, "BiggerData", BiggerData)
[1] TRUE
dbListTables(con1)
[1] "BiggerData" "Iris"
```

Dropping a table from SQLite is possible from R. Notice that we have two tables listed in our database. `BiggerData` is a large table, so it is not instant. Still, the operation is fairly quick. SQLite does not reduce the size of the database file on your hard disk as a result of this operation. Now, it does internally free up that space, so it needs not grow again. However, it is not free to use outside SQLite. In SQL parlance, what is needed is a `VACUUM` of the database. This procedure could require up to twice the size of the file, and again, the space has been released for new use—just not outside the database. Either way, this seems, to us, better done outside R.

```
dbRemoveTable(con1, "BiggerData")
[1] TRUE
dbListTables(con1)
[1] "Iris"
```

Our final bit of code shuts down our connection to our database. The function call is `dbDisconnect()`, and we call that on our connection, `con1`. A successful call returns `TRUE`, and we also see with `dbIsValid()` that it is indeed shut down. We also remove some unneeded data before we move on to the next database.

```
dbDisconnect(con1)
[1] TRUE
dbIsValid(con1)
[1] FALSE
rm(diris_lite1, diris_lite2, diris_lite3, diris_lite4)
```

We said that our space was not freed up. Since this is invented data, to free up your system, simply use Windows Explorer after we are done to delete the `LiteDB.db` file in your `C:/sqlite` folder. It bears repeating that database management should not be done through R. Rather, R can natively store some data if necessary or access subsets of data for analysis.

PostgreSQL

PostgreSQL is both open source and popular (ranking fifth of all databases according to some sources). Highly advanced, this database has over 30 types of data it understands internally. We recommend *Beginning Databases with PostgreSQL* by Neil Matthew and Richard Stones (Apress, 2005). As with SQLite, we walk you through installing a practice version on your local machine. Our goal is to get you using R on data inside this database as soon as possible. More-advanced database operations most likely ought not to be done in R.

This format is recommended when data integrity, reliability, and a need for various levels of user access may be useful. On the other hand, if you just need fast read speed for more static yet large data, this might not be the best choice. Of course, we admit that often the choice of a database has much less to do with what we want and more to do with where data is found. A warning about PostgreSQL is that it does have users, and these users have certain privileges. For our example, we have one superuser. From a security standpoint, this is unwise, and from a real-life perspective, a database owner is unlikely to grant analysts such privileges. If you intend to be working extensively in this environment and want more practice, an easy way to get it is to create more levels of users than we do in our brief installation. Then, practice accessing your database by using those usernames and passwords in R.

Installing PostgreSQL on Windows

Visit the PostgreSQL website (`www.postgresql.org/download/windows/`) and choose one of the installer options. We chose the latest stable release (not a beta) for 64-bit architecture. After downloading, you should have a file named something like `postgresql-9.5.4-windows-x64.exe`, which is your installation program. Of course, your version number likely is higher than 9.5.4, as that is our edition as of the writing of this chapter.

Go ahead and install the program, clicking the Next button as needed, and accepting the default settings for the installation directory. The installer asks for a password for the database superuser named `postgres`, which we set to `advancedr`. If you select a different password, please be sure to record it securely. Incidentally, this also creates a database with the same name as the superuser. PostgreSQL defaults to port 5432, and if that is not showing as the default option, you likely have another version of PostgreSQL living on your computer. While we configure the port in R, we highly recommend at first practice to go with as many defaults as possible. Clicking Next a few more times, accepting the defaults, should lead you to the installation event. You may be asked on the final screen about Stack Builder; it is not required, and you may uncheck that option before clicking Finish.

At this point, PostgreSQL's installation is over, and it is running on your computer. If you are familiar with this database, you could add data to it directly (and likely would not have needed this primer). We add data via R and quickly get to accessing subsets of that data.

PostgreSQL and R

Our first task is connecting to our database. This uses the `DBI` package library and builds a link to our database. In the case of PostgreSQL, we need a username and password. We also need to tell the link the location of our database as well as the type of database. This database is a server, and we do not need to tell it the file location so much as the host location and the port. We call this second connection `con2` and then use the `dbGetInfo()` function from the `RPostgreSQL` library to test whether it started correctly. Ours returns with quite a bit of information, so we are ready to go:

```
drv <- dbDriver("PostgreSQL")
con2 <- dbConnect(drv, dbname = "postgres", host = "localhost",
+                  port = 5432, user = "postgres", password = "advancedr")
dbGetInfo(con2)
$host
[1] "localhost"

$port
[1] "5432"

$user
[1] "postgres"

$dbname
[1] "postgres"

$serverVersion
[1] "9.5.4"

$protocolVersion
[1] 3

$backendPId
[1] 14220

$rsId
list()
```

It may be worth mentioning that we included the password directly in the code in plain text for `dbConnect()` this time. It might not be the most secure methodology, and care should be taken (particularly in this case, since this is our superuser). All the same, our connection is working, it is properly connected to port `5432` (while this is the default port for this database, it is not always this port), and our host is `localhost`. Should you connect to a database online, you would change the address to that database. Now that our connection is working, we see which tables are in the database and add the familiar iris data set:

```
dbListTables(con2)
character(0)
diris <- as.data.table(iris)
```

```
dbWriteTable(con2, "iris", diris, overwrite = TRUE)
[1] TRUE
dbListTables(con2)
[1] "iris"
dbListFields(con2, "iris")
[1] "row.names"    "Sepal.Length" "Sepal.Width"  "Petal.Length"  "Petal.Width"  "Species"
```

A nice feature of these functions is they do what they say. All the same, we draw your attention to the `dbWriteTable()` function. Although writing is not our main focus, this function takes both an `overwrite` and an `append` option set to `FALSE` or `TRUE`. Both are convenient in the right situation. We turn our attention to reading from this database, and confirm that we successfully pull into R data from PostgreSQL with a visual inspection of our data:

```
diris_gre1 <- dbReadTable(con2, "iris")
diris_gre1 <- as.data.table(diris_gre1)
diris_gre1
     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
 ---
148:          6.5         3.0          5.2         2.0 virginica
149:          6.2         3.4          5.4         2.3 virginica
150:          5.9         3.0          5.1         1.8 virginica
```

A basic SQL query `SELECT`s specific columns, `FROM` a specific table, `WHERE` specific rows of specific columns have a certain feature. We use `dbSendQuery()` to send our query to and execute it in PostgreSQL. Then, we use `dbFetch()` to get our results into R. Notice that when we fetch, we select only the first three rows that have this feature. At the end, we use `dbClearResult()` to clear the buffer in PostgreSQL in preparation for future queries.

```
query <- dbSendQuery(con2, statement = 'SELECT * FROM iris WHERE "Petal.Length" > 2')
diris_gre2 <- dbFetch(query, n=3)
head(diris_gre2)
   row.names Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1         51          7.0         3.2          4.7         1.4 versicolor
2         52          6.4         3.2          4.5         1.5 versicolor
3         53          6.9         3.1          4.9         1.5 versicolor
dbClearResult(query)
[1] TRUE
```

This is highly similar to what we did earlier with SQLite, because both databases are SQL databases that store tabular data. Alternately, if we needed only one column of data, we would `SELECT` that specific column rather than using a wildcard value:

```
query <- dbSendQuery(con2, 'SELECT "Petal.Length" FROM iris')
diris_gre3 <- dbFetch(query, n=6)
head(diris_gre3)
  Petal.Length
1          1.4
2          1.4
```

```
3          1.3
4          1.5
5          1.4
6          1.7
dbClearResult(query)
[1] TRUE
```

The next query returns both `Sepal.Length` and `Species` columns from our `iris` data set, where the sepal width does not equal `2.2` (an arbitrary number). Of note is that this returns only `147` rows instead of the full `150`, and of course only two columns instead of five. Scaling this idea up, you readily see how big data might be broken into relevant chunks for analysis. We changed from `query` to `qu` simply to save a few characters, to improve the readability of the line in the query text.

```
qu <- dbSendQuery(con2, 'SELECT "Sepal.Length", "Species" FROM iris WHERE "Sepal.Width" <>
2.2')
diris_gre5 <- dbFetch(qu, n=-1)
head(diris_gre5)
  Sepal.Length Species
1          5.1  setosa
2          4.9  setosa
3          4.7  setosa
4          4.6  setosa
5          5.0  setosa
6          5.4  setosa
dbClearResult(qu)
[1] TRUE
```

The purpose perhaps of PostgreSQL is to allow for data larger than RAM might physically allow. It can be read in chunks as we've demonstrated, or we can push large amounts of data from R to free up memory. Our next example comes with a choice. If you have enough available free space in your system RAM, go ahead and do the example of 200 million rows as written. Otherwise, either change the number of variables in `rnorm()` to something lower, or use our code online to run the smaller example of 70 million rows.

This code creates 200 million entirely made-up data observations with a mean of 1 and a default standard deviation of 1. As you see, it is about 1.6 gigabytes in size. Certainly, data can be larger, but this might be good enough as a test case to see how it works on your system. It likely takes around 5 minutes to run, so be prepared for a bit of a wait. If you have already run this in an earlier step, there is no need to repeat it.

```
BiggerData <- rnorm(200000000, 1)
BiggerData <-as.data.table(BiggerData)

dbWriteTable(con2, "BiggerData", BiggerData)
[1] TRUE
dbListTables(con2)
[1] "iris"       "BiggerData"
```

Once again, while we are (at least with superuser privileges) able to `DROP` a table, it does not release space to us in PostgreSQL (and hence on our hard drive). Also once again, we recommend saving `VACUUM` commands for actual database management (and thus recommend using SQL on the database directly rather than in R).

```
dbListTables(con2)
[1] "iris"       "BiggerData"
dbRemoveTable(con2, "BiggerData")
[1] TRUE
dbListTables(con2)
[1] "iris"
```

To close this connection, there are two steps rather than one, because we must disconnect from PostgreSQL and unload our driver that allows `DBI` to make that connection. If successful, they will both return `TRUE`.

```
dbDisconnect(con2)
[1] TRUE
dbUnloadDriver(drv)
[1] TRUE
```

This concludes our demonstration of both PostgreSQL and tabular databases. The real way to make the knowledge you now have more powerful is to delve further into SQL syntax to use `dbSendQuery()` on ever more exotic quests. For now, though, a small-enough quantity of data ought to be readable into R that some analytics can happen. We turn our attention to a different sort of database.

MongoDB

As mentioned earlier, *MongoDB* is different. This is a document store database, which is a type of nonrelational database. This avoids tables for documents called objects. These documents are binary versions of JavaScript Object Notation (JSON); since they are binary versions, MongoDB calls them BSON objects. This type of database is used for data that may be semistructured or unstructured. This tends to be closer to the concept of a *data lake* (an increasingly popular buzzword in data science), which suggests that diverse types of data may need to be stored and accessed.

One powerful feature of JSON and semistructured data is the ability to collect all the information in one place for one particular instance. For example, suppose we had all patient records in one place. Rather than dozens of tables in which each patient occupies a row, and each table is something along the lines of Address or Diagnosis, we instead store all records for one patient in one blob. Of course, most patients have an address, but perhaps not all need three types of blood panels in their diagnosis section. This is semistructured data. On the other hand, if you are simply collecting data from the Web about a favorite company—including news articles, social media posts or likes, and stock market performance figures—there may be almost no similarity between elements of that data. Regardless, MongoDB can store it, and we can access it.

Installing MongoDB on Windows

A visit to the MongoDB website (`www.mongodb.com/download-center`) shows an option to download the 64-bit version of MongoDB with SSL support (version 3.2.8 as of this writing). See Figure 10-1. Once that is downloaded, go ahead and start the installation process. Select Next, read and agree to the license documentation, and proceed. We used the complete installation and then clicked the Install button. A short time later, please click Finish.

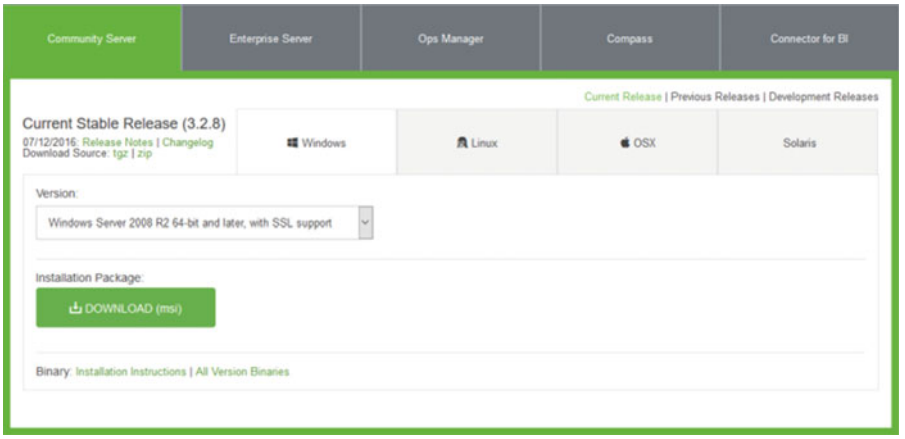


Figure 10-1. *Downloading MongoDB*

We also need to download the sample database for MongoDB. We will use the restaurants data set available at MongoDB's Import Example Dataset page at https://docs.mongodb.com/getting-started/shell/import-data/ under step 1 (`primer-dataset.json`). We will also use `Ch10_MOCK_DATA.json` as another data set. Both are also available as part of the code packet for this chapter from this book's site. Go ahead and download both files, and store them in `C:\Program Files\MongoDB\Server\3.2\bin`. Keep in mind that you may be using version 3.3 or later, so be sure to change the version number in this file path as needed. With that, MongoDB is installed, and we have our data sets. Figure 10-2 shows the folder.

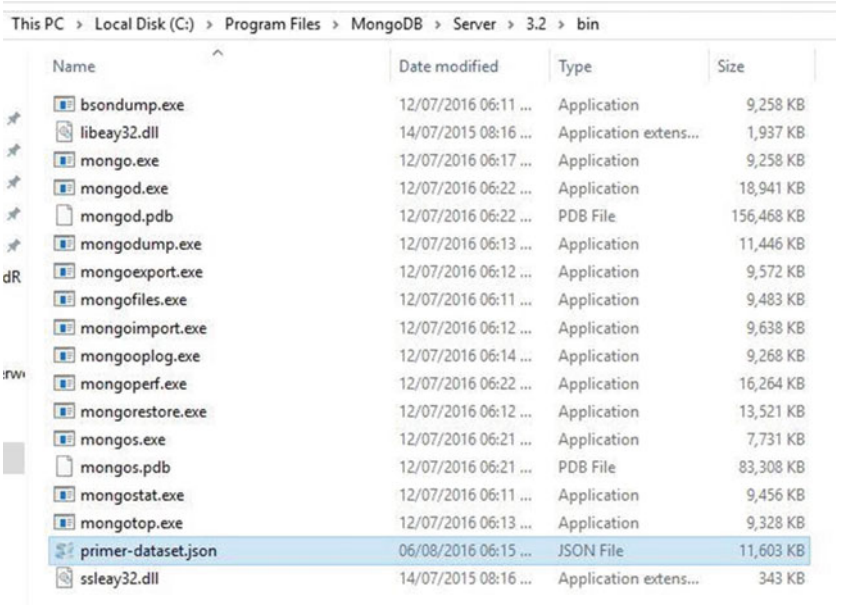


Figure 10-2. *The MongoDB\Server\3.2\bin folder*

Now press the Windows key, and while holding it down, then press the R key. This opens the Run dialog box. Type **cmd** and then press Enter/Return. There needs to be a data directory for your database. By default, this is in the main file path (for example, usually the C drive for Windows). In the command prompt, type `md \data\db` to make the default directory. Note the space between `md` and `\data\db`. Next, still in the command prompt, type `cd C:\Program Files\MongoDB\Server\3.2\bin` and hit Enter/Return. Should version 3.3 release before the printing of this book, change the 3.2 in the preceding file path to 3.3. From here, type `mongod.exe` and hit Enter/Return. If all goes well, the last line should be `waiting for connections on port 27017`. Keep this window open (otherwise, our server is not serving files).

Next, again keeping the first console open, press and hold the Windows key and press the letter key to open a second Run dialog box. This time, most likely `cmd` is already filled in, so press Enter/Return to open a second command prompt. In the command prompt, again type `cd C:\Program Files\MongoDB\Server\3.2\bin` and hit Enter/Return. We now use MongoDB itself to import these two data sets. Run the code shown in bold, and the following output shows the results:

```
mongoimport --db test --collection restaurants --drop --file primer-dataset.json
2016-09-25T11:11:24    connected to: localhost
2016-09-25T11:11:24    dropping: test.restaurants
2016-09-25T11:11:26    imported 25359 documents

mongoimport --db test --collection mock --drop --file Ch10_MOCK_DATA.json
2016-09-25T11:19:45    connected to: localhost
2016-09-25T11:19:45    dropping: test.mock
2016-09-25T11:19:45    imported 1000 documents
```

Note that this database has a database named `test`, and inside that database there are now two collections. These collections, namely `restaurants` and `mock`, were created by importing JSON files. Had those two collections existed already in our `test` database, then the `--drop` command would have purged them. It is somewhat standard to use a database named `test` or `temp` as a way of signaling that experimentation is expected in such a place.

Go ahead and close this second command prompt window (be careful—you must keep the first window, which should be giving you server `COMMAND` and `NETWORK` feedback, open). After closing just this second window, we turn our attention to `R`.

MongoDB and R

MongoDB does have the option to have usernames, passwords, and multiple databases. As we have not set up any of these, we go ahead and simply connect to our local server on our third connection, `con3`. Unlike our first two connections, this uses a new command, `mongoDbConnect()` from the `RMongo` package . Also, notice that this calls for a host to be the localhost, and the port is specified as well. If you are connecting to a MongoDB instance over the Web, be sure to identify the IP address as well as its port. Of course, also find out from the administrator the name of the database you wish to access. Finally, find out whether you would have only read permissions or more. It does not do to experiment on other people's databases!

```
con3<-mongoDbConnect("test", host = "127.0.0.1", port=27017)
```

Should authentication be required, you need to run a second command. Type and run `dbAuthenticate(con3, username, password)` into your R session. A word of caution is needed here: depending on the version of MongoDB being accessed, the version of Java installed on your local machine, and the authentication protocol set up by the database administrator, this functionality may or may not work. We realize that this is a vague statement. As of the writing of this book, authentication seems possible, but there seem to be signs that this is not always true for all users.

To add to the bad news, at this time, the function call `dbShowCollections()` does not work on the most recent versions of MongoDB. If you are connecting to a mature database, chances are it may be using an older version, and you may be able to see the collections. The `RMongo` package is approximately three years old, and we have hopes for an update. For now, we simply provide the heartless wisdom that connecting to databases and data munging, in general, is rarely easy. Fortunately, we know that our collections are stored in `restaurants and mock`.

Thus, we may proceed to access some data from our collections to confirm that we can, in fact, read data from MongoDB into R. Notice that the function call `dbGetQuery()` takes at least three arguments. First, it requires the connection, and second, it requires a text string that has `collection` information. The third argument is the search terms used to access our data. In this first case, we do not define any search terms and thus merely have the wrapper of {} for that formal. The last two arguments are optional. The `0` tells us we are starting at the beginning of the list, and the `8` says we want the next eight entries only. In other words, we are skipping no entries, and we are limiting the ones returned to eight. We have edited the final output for readability and show just a portion of the `View()` in Figure 10-3.

name	cuisine
Kosher Island	Jewish/Kosher
Brunos On The Boulevard	American
C & C Catering Service	American
Wild Asia	American
Riviera Caterer	American
Wendy'S	Hamburgers
Morris Park Bake Shop	Bakery
Taste The Tropics Ice Cream	Ice Cream, Gelato, Yogurt, Ices

Figure 10-3. *Part of the results of the View(output1) command*

```
output1 <- dbGetQuery(con3, 'restaurants' , "{}", 0,8 )
is.data.frame(output1)
[1] TRUE
names(output1)
[1] "address"  "restaurant_id"  "name"  "cuisine"      "X_id"  "borough"      "grades"
output1[1,]

address
{ "building" : "2206" , "coord" : [ -74.1377286 , 40.6119572] , "street" : "Victory
Boulevard" , "zipcode" : "10314"}

restaurant_id   name           cuisine        X_id                           borough
40356442        Kosher Island  Jewish/Kosher  57e7f9a275bec64c417b9ab2       Staten Island


                                    grades
[ { "date" : { "$date" : "2014-10-06T00:00:00.000Z"} , "grade" : "A" , "score" : 9} ,

 { "date" : { "$date" : "2014-05-20T00:00:00.000Z"} , "grade" : "A" , "score" : 12} ,
```

```
 { "date" : { "$date" : "2013-04-04T00:00:00.000Z"} , "grade" : "A" , "score" : 12} ,

 { "date" : { "$date" : "2012-01-24T00:00:00.000Z"} , "grade" : "A" , "score" : 9}]
View(output1)
```

This data seems to contain information about locations, types of food, and names of restaurants. Using that knowledge, we refine our search criteria to types of `American` cuisine. This is interesting because what we do is remove the start and limit arguments, and adjust the query criteria. Here is where data hygiene becomes so crucial. Naively, we attempted `dbGetQuery(con3, 'restaurants' , '{"cuisine": "American"}')` at first. This did not work. Because we already had to rewrite this section because our former favorite package for this database was removed from CRAN, this seemed somber. Many things were tried; many things failed. In the end, a careful inspection of the data set in the raw downloaded JSON file revealed that for whatever reason, there is a space at the end of *American*. Not the other food types, mind you, just that one. If we had selected any other option, we would not have had this story to tell. The lesson learned that we want to share is that this process takes time. Expect things to go not so well. Keep persevering, and the data will come eventually.

After correcting our query, we can get the results. Rather than simply view those results, we decided to put the restaurant names into a word cloud, to see whether we could detect a pattern. Be sure to note the space after our cuisine type! Also note that if you search for a *Bakery*, this is not necessary. We show the results of the word cloud in Figure 10-4.

Figure 10-4. *A word cloud comprising the names of 1,000 American cuisine restaurants*

```
output2 <- dbGetQuery(con3, 'restaurants' , '{"cuisine": "American "}')
output2 <- as.data.table(output2)
rNames <- Corpus(VectorSource(output2$name))
wordcloud(rNames, max.words = 20)
```

Our other, `mock` data came with an ID value already, and MongoDB gives each document or blob an `id`. Thus, to avoid name conflicts, `X_id` was made by MongoDB. This is a rather important point; most SQL or NoSQL databases tend to have some reserved names. While we did our best to ignorey that in this chapter, if you intend to store data yourself on such a database rather than simply read it into R, we recommend reading more on the topic and having healthy caution. We also formatted this output a bit to clean it up.

```
dbGetQuery(con3, 'mock', '{}', 0,1)

gender
   Male

financial
{ "account1" : "DO56 T7LH 5884 0956 8784 3936 1833" ,

 "account2" : "197VPF5XPA13ngPwAmSiWsSMoUUPoWFQpa"}

last_name       X_id                          id       ip_address     first_name
Lawson          57e7f92175bec64c417b96ca      1        255.61.75.176  Jesse

diagnoses
Manic affective disorder, recurrent episode, moderate

email
jlawson0@faux.com
```

As you can see from our first bit of mock data, we seem to have patient information. We hasten to assure you that this is all imaginary data. Of note in blobs is that while this first one did, in fact, have all known bits of data, there is no reason for each item to be present. Thus, if only one financial account were available, we would simply not include the second account. Unlike tabular data, which requires NA entries of some sort, in semistructured data, we can simply not include elements that are unknown. These data are thus highly flexible (and more efficient in storage, because unknowns are not included at all).

The downside is that searching through such unstructured data can be fairly difficult without a strong knowledge of the data set and naming conventions. All the same, we can find all documents in our collection that have a specific feature in common. The function call `dbGetQuery()` can take many things in the third formal. Here we have just passed it a list that finds only documents in which the gender is `male`. Again, additional formals include `skip`, which skips the first *x* rows, and `limit`, which limits the results to an upper cap.

```
mfound<-mongo.find.all(con3, "test.ch10data", list(gender="Male"))
```

For readers more familiar with MongoDB queries, the third formal can, in fact, contain BSON query language directly. To learn more about queries in MongoDB, we recommend *The Definitive Guide to MongoDB, Third Edition* by David Hows et al. (Apress, 2015), particularly Chapter 8. For now, we take our records indicating male gender and pull out the first names to do a bit of analysis on semistructured data. The types of data stored in such blobs are often less suited to traditional statistical tests, and perhaps more related to modern analytics. For simplicity's sake, after extracting the male gender blobs, we access the first-name component and store that in a `corpus` called `fNames`. From there, we build a word cloud on the first 20, as shown in Figure 10-5.

```
output3 <- dbGetQuery(con3, 'mock', '{"gender": "Male"}')
fNames <- Corpus(VectorSource(output3$first_name))
wordcloud(fNames, max.words = 20)
```

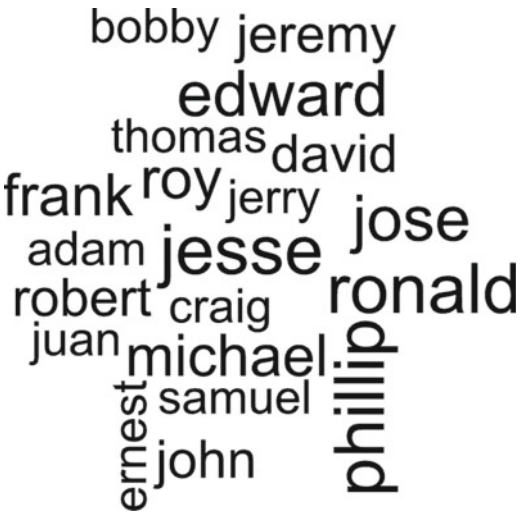


Figure 10-5. *Word cloud of first names, where gender = male*

Our final bit of code shuts down our connection to our database. The function call is `dbDisconnect()`, and we call that on our third connection, `con3`. This call both disconnects and frees up client and server resources:

```
dbDisconnect(con3)
```

Quite a bit more could be said about unstructured documents and the exciting possibilities of this type of data. The ability to store and then access chaotic information becomes ever more routine. This is an area where we anticipate much room for growth, as fuzzy matching allows machines to sort through quickly and return near matches—perhaps a topic for other authors, and certainly a topic for another book. The last step to take in this section is to close the command-prompt window that is still open.

Summary

This chapter gave you direct access to three databases with R. While many of the features are not designed for curating data, R nevertheless provides a quick way to get a handle on big data. Such data sets seem to be becoming ever more common, and while it may not be feasible for researchers to be familiar with every database, that is not required. Using R, it is entirely possible to pull out a subset of data that contains all items of interest and is small enough to fit into working memory. Skilled users of R, such as this book's readers, can prune the data in R itself further, and from there perform desired analytics. Table 10-1 provides a summary of the key functions in this chapter.

Table 10-1. *Key Functions Described in This Chapter*

Function	What It Does
`dbConnect()`	Uses the DBI package plus a driver to connect to a SQLite or PostgreSQL database.
`dbDriver()`	Driver needed for PostgreSQL (and others).
`dbIsValid()`	Tests a SQLite connection for validity and returns a Boolean.
`dbWriteTable()`	Writes data from R to SQLite or PostgreSQL (you must be a user with write privileges).
`dbListTables()`	Lists all tables in a SQLite database.
`dbListFields()`	Lists all fields (column names) in a SQLite database.
`dbSendQuery()`	Sends an SQL query to a SQLite or PostgreSQL database and executes the query there.
`dbFetch()`	Fetches the results from the last sent query in SQLite or PostgreSQL.
`dbClearResult()`	Sets SQLite or PostgreSQL back to neutral for the next query.
`dbRemoveTable()`	Drops a table from SQLite or PostgreSQL (user must have privileges); does not `VACUUM`.
`dbDisconnect()`	Closes the DBI package connection to SQLite or PostgreSQL.
`dbUnloadDriver()`	Unloads the PostgreSQL (and other) drivers.
`mongoDbConnect()`	Creates a connection to a MongoDB server and a specific database on that server.
`dbInsertDocument()`	Inserts a record into a MongoDB collection; takes second formal collection.
`dbGetQuery()`	Finds all blobs that match a query; this has several possibilities we only briefly explored.
`dbDisconnect()`	Destroys a MongoDB connection.

CHAPTER 11

Getting a Cloud

Depending on your needs and uses for R, it can be convenient to have a reasonable amount of memory and processor capability. Often this power is necessary only on occasion, and it may not be cost-effective to own the hardware. This is where hosting R on the cloud may be helpful. Cloud instances bring on-demand resources that are readily scaled up or down as each situation requires. These days, there are several tolerable outfits that provide such services at very reasonable prices. The challenge we face is threefold, and we walk through those steps over the next three chapters. We need to get a cloud, we need to administer our cloud's operating system, and we need to do some fun things with R.

This chapter focuses on getting a cloud. To do that, we need to choose a provider, start up a compute instance in a location we will not ever physically access, and successfully connect to our instance. We are going to make some assumptions along the way, and, given the quickly evolving nature of technology, this may well be obsolete even as we write.

Our first claim is that you, our gentle readers, have access to a Windows-based computer and are most familiar with a Windows environment. We do not suppose you have administrative privileges on your local machine, although it requires a network connection. You also need access to a credit card. Although your work in this chapter doesn't incur a cost , a credit card is required. Finally, and apologies for those for whom this is not true, we are going to assume that you have no prior knowledge of clouds, networks, or Unix-flavored operating systems.

Disclaimers

There are a few disclaimers we should make at this point. We are in no way endorsing Amazon Web Services (Setting Up with Amazon AWS EC2, 2016) just as we are in no way endorsing Windows. Nevertheless, AWS is convenient, popular, and, as of this writing, offers a free tier of service for 12 months (`http://aws.amazon.com/free/`). Please be careful with data you place in the cloud; without proper attention to security, it might be easy for that information to find its way to the wrong minds. While we give some brief suggestions on digital safety, these are not enough. The topic of Internet security can and does fill several books; if you have data that should not be faxed or e-mailed, there is much more to be learned before processing that data on a cloud.

With luck, we have not scared you away from this convenient and powerful method of bringing just the right resources to your research and data. We also recommend reading this chapter and the one following first, and dedicating an afternoon or more to the process if you are, like us, cloud novices. We should also mention that we use Google's Chrome browser; your choice of browser may change the screenshots shown in this chapter. Indeed, AWS itself has semiregular layout changes as new features are added or old functionalities merge. Nevertheless, what you see should look quite close to what we show.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_11

Starting Amazon Web Services

Very few people are polymaths, and living in the cloud can be a bit difficult conceptually. In our experience, being good at mathematical programming does not instantly translate into cloud expertise. Fortunately, the system is fairly well designed to start up quickly after some legwork is done. What you want is access to an Elastic Compute Cloud (EC2) instance. EC2 is a virtual computer that can be customized to have whatever processing power and memory combination you may wish. That is an overly bold statement. However, at the time of this writing, anywhere from 1 processor at 2.4 GHz and 0.5 GiB of memory, to 40 processors at 2.4 GHz and 160 GiB of memory, can constitute a single computer instance.

The first step we take is visiting the Amazon Web Services site (`http://aws.amazon.com/free/`) and getting a new account, as shown in Figure 11-1. Follow the steps to create a new account, being sure to generate a secure password. This step requires both a credit card and a phone number. Again, we use the Google Chrome browser, and your browser's view may differ from ours on occasion.

Figure 11-1. *Creating a new AWS account*

The account you have just created is your top-level account. The recommendation is to create a subaccount that does not have the full rights this one does. In fact, Amazon goes to some pains to convince you to use identity and access management (IAM) to create a secondary account. If you are just a single user who never needs someone else to access this, a secondary account is still beneficial. If you are ever going to have other users, it is a no-brainer decision. From adding a graduate student, to adding a colleague, to adding employees or consultants, it can be helpful to not give away the master key to the castle. Do not be shocked by the plethora of options available; simply scroll down to the Security and Identity portion of AWS and select IAMS from near the end of the middle column, as shown in Figure 11-2. Depending on your browser, IAM may have a key icon or may have no icon at all. There are enough options present that pressing Ctrl+F and then typing **Identity & Access Management** into the search box may help highlight the text for IAMS. Do not be shy about searching, clicking, and exploring to find it. You can't hurt anything, and Amazon updates the interface on occasion.

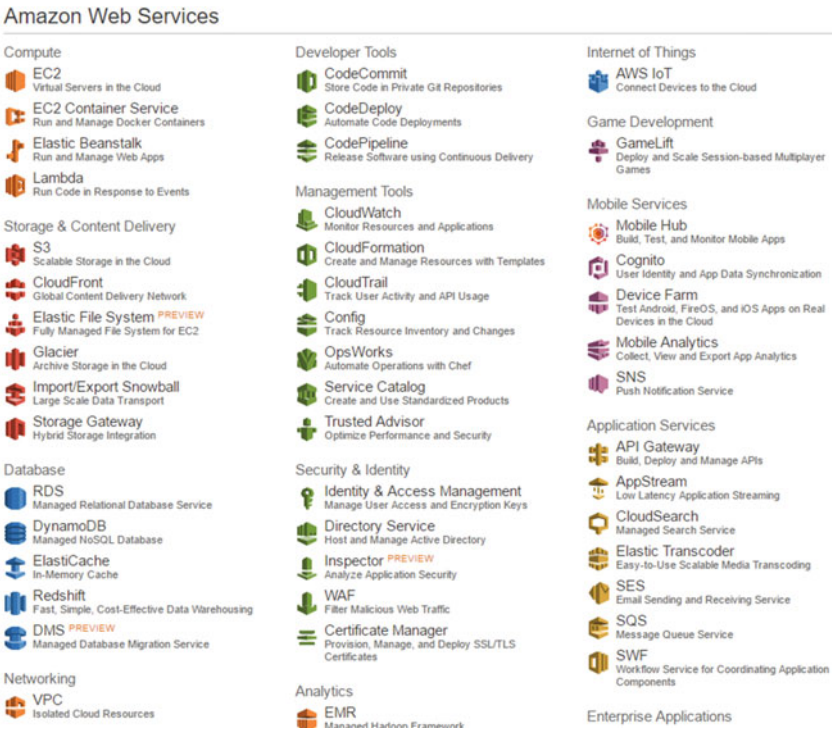


Figure 11-2. *The AWS management console with highlighted boxes for the three parts we use in this chapter*

Inside IAMS, you need to create both a new group and your first user. Groups are convenient because they allow you to create multiple users who have only certain privileges. On the left side of the screen, select the Groups option and create a new group. We called ours *AdvancedR*, although yours might be more sensibly named CloudAdmin or something similar. When asked which policy, we recommend AmazonEC2FullAccess, as shown in Figure 11-3.

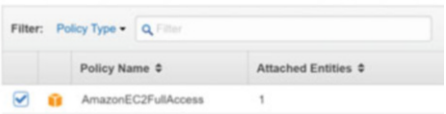


Figure 11-3. *IAMS group policy selection—your view may differ*

Next, we create a user. While following directions on the establishment of the user, we recommend using your first name so that you see that account as *you* and using a different password from your AWS account. Start by selecting the Users tab on the left side. Once there, create your username. Also, this is the stage where, if you are going to have a consultant help you build your cloud, you can create a second user account for them. Be sure the settings are as in Figure 11-4, including deselecting access-key generation.

Figure 11-4. *Creating a user on the AWS IAMS*

After creating your user(s), select your username(s) by clicking the check box (not clicking the username) and the User Actions tab. From there, add them to your new group. Also from there, select your user(s). From the User Actions drop-down list, select Manage Passwords and then assign a custom password or autogenerate a password. See Figure 11-5.

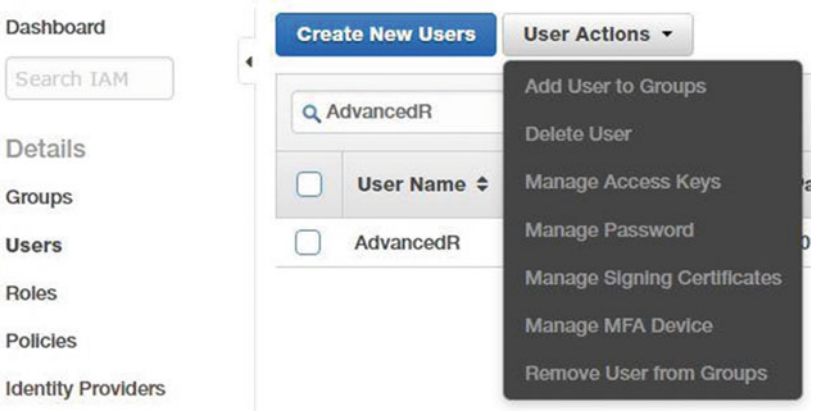


Figure 11-5. *User Actions tab and drop-down box*

When you created your account initially, AWS gave you an account identification number. To see this number and link again, click the IAM Dashboard at the top left, as shown in Figure 11-5. The Dashboard has an IAM users sign-in link as well as an option to customize that to a name. Sign out of AWS. Go to your AWS address that follows this pattern: `https://YOUR_ID_HERE.signin.aws.amazon.com/console`; sign in using your new username and password.

Taking a look at Figure 11-2 again, notice in the top-left corner the link for EC2. Click this link; at the top-right, you can select a region such as Oregon or Sydney. Different regions tend to have variations in cost, and of course, the closer you are geographically to a region, the more likely network upload and download speeds are efficient. As is often the case, there are trade-offs to consider; we recommend selecting the region nearest to you. One warning is that after you select a region, some of what we do is regionally specific. In particular, key pairs (used to access your cloud server) tie to particular regions.

Once you have your region selected, it is time to create a key pair. On the left side is a menu list, and under Network & Security is the Key Pairs link. Select that link, and then click the Create Key Pair button. Give your key pair a name. If you are likely to need cloud instances in different regions, Amazon's advice to

add your region name to the key pair name is sound. Your browser should automatically download the key pair file. Remember where this is saved and keep it safe; this is your only chance to download this file. We use it later in the next section, but for now, we move on to creating a virtual private cloud (VPC). Return to the home screen of Figure 11-2.

One last look at Figure 11-2 shows that in the first column, near the end, under Networking, is VPC. There are several options here; we go with the simplest, one with a single public subnet. Select the Start VPC Wizard and give your VPC a name such as *AdvancedR*. Stick with the default options and then select Create VPC. As you can see from these options, if you were interested in creating a more advanced structure, that is readily done.

Your cloud is not safe without a firewall, which is called a *security group*. We are very cautious with our security group. On the left in the VPC Dashboard, notice the Security tab and the Security Groups link. Click that link and then click the Create Security Group button. Give the group a name tag and a name such as AdvancedR, as well as a description. Be sure to select your VPC from that-drop down list at the end and then click the Yes, Create button.

Now, select your AdvancedR security group and notice the options that are at the end of your browser screen, shown in Figure 11-6. What we are about to do is configure which IP addresses are allowed to visit your future cloud instance. This is, of course, not ironclad; however, IP spoofing is way beyond the scope of this chapter. Certainly, this is a good first step, ensuring that you are the only one who knows your server is there. Be sure to select the Inbound Rules tab, select Edit, and add SSH (22) with TCP (6) protocol; you want to put your IP address for the source. Amazon has links to an IP detector, but a simple Google search of **ip address** returns your public IP address as the top hit. It should be in valid Classless Inter-Domain Routing (CIDR) notation, which means that if your address is `72.18.154.27`, you want to type in **72.18.154.27/32** for Source. While we are here, we create rules for ports 80, 443, and 8787—which are HTTP, HTTPS, Shiny, and RStudio Server, respectively. However, we also lock these to just your IP address. This likely prevents others from accessing your server until you are ready. Go ahead and leave the Outbound Rules to allow all traffic to destination `0.0.0.0/0`.

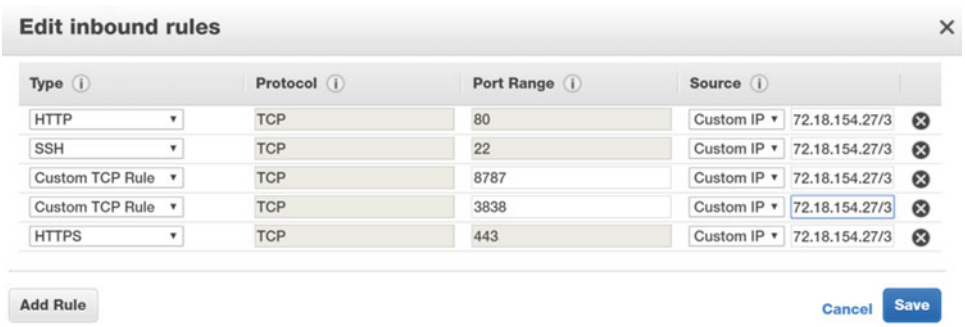


Figure 11-6. *Security group's inbound rules with CIDR notation*

If your IP address changes often, you may need to relax your rules. We recommend doing so as cautiously as possible, and only after you are through with the basic operating system and safety updates we discuss in Chapter 21. Also notice that as of now, if you were to host a web server on your cloud instance, that website would be visible only from your local computer or network. Later, after we have installed a site, we'll go back to this screen and make some edits to allow certain parts of our server to be visible to a wider audience.

While a full lesson on CIDR is beyond the scope of this book, a little knowledge makes sense here. The Internet does not use word addresses such as `www.elkhartgroup.com`. Rather, it uses a series of 32-bit numbers; try typing `72.18.154.27` into your web browser. Computers use binary numbers rather than

base 10; they see our website as `01001000.00010010.10011010.00011011`. That unique address connects us to a particular machine. The `/32` tells AWS that only that machine is allowed; all 32 bits are fixed. Contrastingly, for outbound rules, we need our instance to be able to browse the Web at will. Thus, we use `/0` to let it know that none of the bits is fixed. If you are in a corporate or university environment and want some coworkers to be able to access your instance, talk to your IT department to see what the size of your address space is. If we found our IP addresses changed frequently, we might try `72.18.154.0/24`, which would allow the range of addresses `72.18.154.0-255` (and would nicely include the `.27`).

We turn our attention now to creating an EC2 instance. Select the VPC Dashboard link at the top left, and then click the Launch EC2 Instances button. Select the Ubuntu server that is free tier eligible, as shown in Figure 11-7.

Figure 11-7. *Ubuntu server selection for starting your first EC2 instance*

Next, select the `t2.micro` type. On the next screen, be sure to select your VPC as the Network setting, and set the Auto-assign Public IP to Enable. You always have the option to Review and Launch or to go through the entire start wizard by selecting Next. Keep selecting Next, and, in the following steps, there is no need to change the storage settings for Root volume; as of this writing, up to 30 GiB are free. For tags, a good name is AdvancedR again (admittedly, the name is getting a shade overused, yet it continues to work). Also, for the security group, be sure to select an existing security group as well as your AdvancedR group before clicking Review and Launch. From there, launch!

Your last step is to select your key pair. Remember, we already downloaded this earlier. Be sure you have that file, and be sure to have the settings of choosing that existing key pair and selecting your AdvancedR key pair. Next, select the acknowledgment check box. Finally, click the Launch Instances button, as in Figure 11-8.

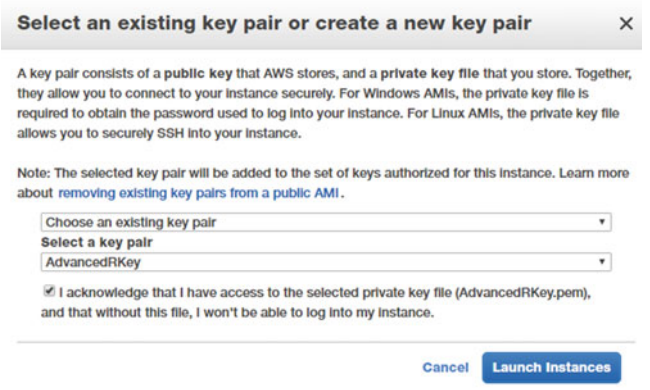


Figure 11-8. *Instance key pair settings and final launch*

From here you may view your instances, which take you to the EC2 dashboard. It does take some time for the instance to spin up, so, while we are waiting, we'll go ahead and meet you in the next section.

Accessing Your Instance's Command Line

You have an instance on AWS, and it is starting and waiting for you to connect to it. Before that can happen, there are some steps to take on Windows.

You need to download both PuTTY (Tatham, 2016) and PuTTYgen, and you want to download the installer for Windows for all files except PuTTYtel from www.chiark.greenend.org.uk/~sgtatham/putty/download.html. Once that is done, run the file to install. If your local machine does not grant you the privileges to install, you may download each of these one at a time.

It is the PuTTYgen we want as well as the *.pem file we downloaded as our key pair file earlier. You want the settings as in Figure 11-9. In particular, as you are loading your existing private-key file, you want to change to All Files (*.*) and open your key pair file. You also want SSH-2 RSA as the parameter setting. Once you click to open your key pair file, you see some information appear in the generator. It is safe to ignore all of it and click Save Private Key. PuTTYgen asks if you are willing to save without a passphrase, and after agreeing to that, you may now save the file as a *.ppk file. Keep this file safe! It is your access pass to your server instance.

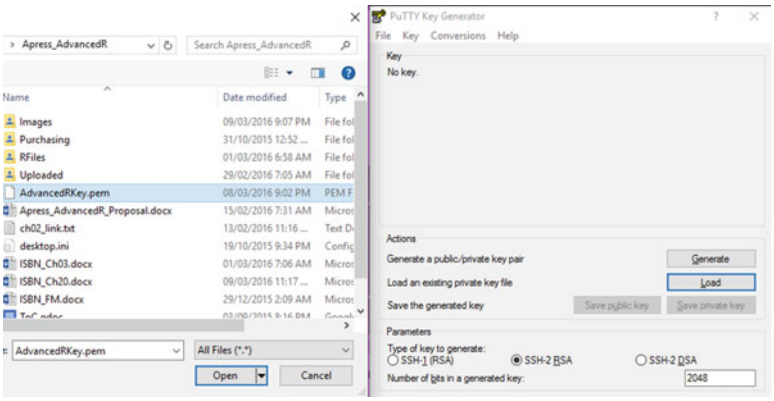


Figure 11-9. *PuTTYgen key generator*

We may now connect to our instance. Going back to the EC2 Dashboard, the instance should be running, and there is some information we need from Figure 11-10. The Public IP address is necessary, and of course, the fact that it is running. Be sure to copy the address for your instance.

Figure 11-10. *EC2 Dashboard view of our instance with Public IP address and the Actions drop-down menu*

Click the Actions drop-down menu, and under Instance Settings go to Get System Log. At the end of the log file, there should be the server's fingerprint, as shown in Figure 11-11. The first time we connect to our server, we want to check that we successfully connected to the server we were expecting rather than a fake. For safety, remember this or keep that window open and be ready to make a comparison.

```
ec2: 2048 0e:e9:14:3f:a7:94:e9:f6:95:33:b7:a0:20:e3:15:48  root@ip-10-0-0-244 (RSA)
ec2: -----END SSH HOST KEY FINGERPRINTS-----
ec2: #############################################################
```

Figure 11-11. *Server fingerprint (RSA)*

Now we finally connect to our server for the first time. Go ahead and start PuTTY. On the navigation menu on the left, choose the SSH option, then the Auth option, and browse for your *.ppk key, as pictured in Figure 11-12.

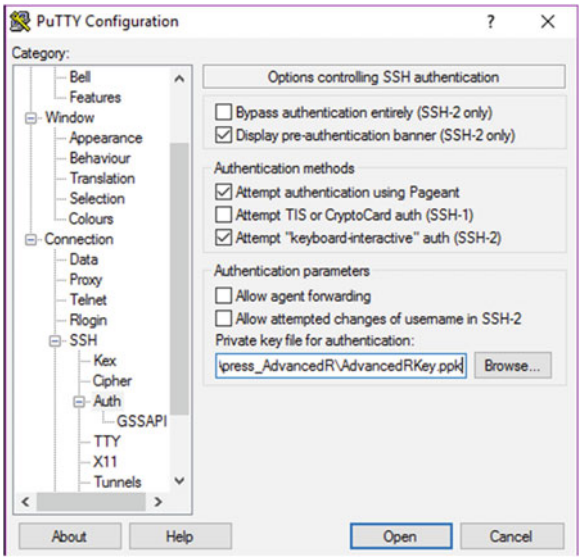


Figure 11-12. *PuTTY configuration for *.ppk key file*

Go back to Session in the navigation menu on the left, and enter **ubuntu@54.201.117.64** or more generally **ubuntu@YOUR_EC2_PublicIP_HERE**. Make sure that the Port is set to 22 and that the Connection type is SSH. We recommend saving your session by writing AdvancedR into the Saved Sessions text and clicking Save before proceeding. This makes it easy to access your `t2.micro` instance in the future. Go ahead and open the connection, and PuTTY gives you a security alert, as in Figure 11-13; you should compare this favorably to Figure 11-10 before proceeding by selecting Yes. If these two fingerprints do not match, something not good at all is probably afoot.

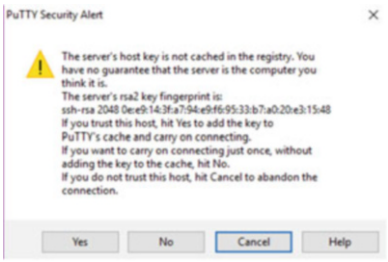


Figure 11-13. *PuTTY security alert for server fingerprint*

You are now connected to your cloud instance, and you are in the Ubuntu command line! As shown in Figure 11-14, you have access to the command line and are now ready to move on to using your server as shown in Chapter 21 if you wish.

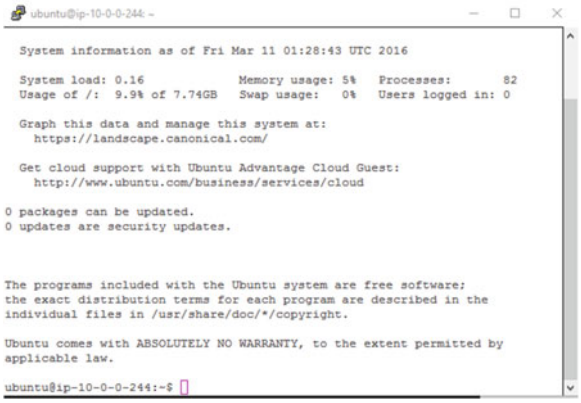


Figure 11-14. *Seeing the command line through PuTTY's eyes*

We wait just a bit before we get to using our server. While we need to take some more steps to get R functional on the cloud, we take on one more section here to allow ourselves the luxury of uploading files to our cloud. There is one command we should teach you before we upload files. It is the `exit` command, and it is simply the word *exit* followed by Enter/return. This closes your PuTTY session.

```
ubuntu@ip-10-0-0-244:~$ exit
```

Uploading Files to Your Instance

We use WinSCP (Prikryl, 2007) to upload files to the cloud. This program may be downloaded from https://winscp.net/eng/download.php (we used the Portable Executables link). We unzipped the files to our desktop and clicked the `WinSCP.exe` files to open our initial window, which we filled in as shown in Figure 11-15. Specifically, we entered our IP address for the hostname, ensured we were set up to connect to port 22, and entered **ubuntu** as the username.

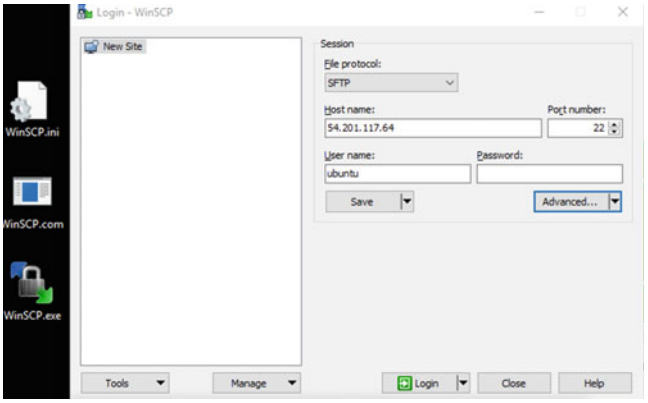


Figure 11-15. *WinSCP start screen with correct settings*

For the password, select Advanced, and in the new window that opens, in the SSH tab and under Authentication, navigate to the same `*.ppk` file you used for PuTTY. We show the results in Figure 11-16. After clicking OK and then Save, you are ready to log in.

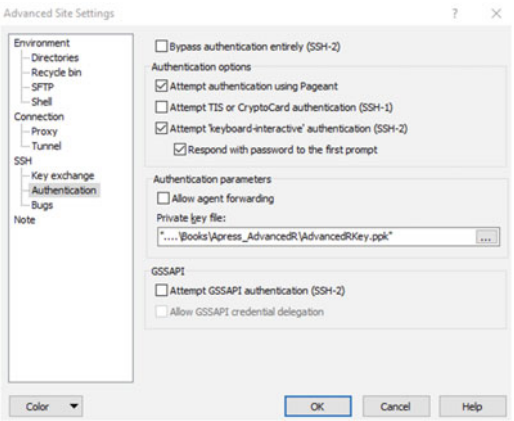


Figure 11-16. *WinSCP Advanced key file screen*

When you saved, you had the option to create a desktop shortcut, which can be quite convenient if you regularly have files to transfer. At your first login, you see your server's key fingerprint. Again, just as with PuTTY, be sure to check that against the fingerprint in Figure 11-10. We go ahead and upload that old Chapter 2 file called `ch02_link.txt` to our `/home/ubuntu` folder on our server. To do this, on the left side navigate to where you are storing the files you downloaded from our online code packet. Then, drag and drop that text file to the right-hand side. Once it has uploaded, we go ahead and disconnect from the session and close out of the program. Remember the file name, though; we use that file in the next chapter.

Final Thoughts

Both PuTTY and WinSCP have versions that work without installation. Thus, even on a computer on which you do not have administrative rights, you can still access your instance and transfer files back and forth. This can be helpful in the corporate or academic environment, where technology services departments often rule local computer rights with an abundance of caution.

Speaking of caution, you should be cautious with your cloud server. While we build things in a reasonably secure fashion, this is not entirely safe. It is likely somewhat safer than e-mail. Often, research data may have either economic or privacy value. In that case, you undoubtedly want to consider some Internet security texts. Proceed at your risk; this is not a book about securing your cloud!

CHAPTER 12

Cloud Ubuntu for Windows Users

As you saw in the last chapter, using the cloud does not always allow for using Windows. This is not entirely true; Windows can certainly be utilized in the cloud. However, this tends to approximately double the cost in high-performance environments, making it impractical for some occasions. Additionally, it is not impossible to use the Ubuntu environment, and many of the same skills advanced R users have translate readily. All in all, using Ubuntu as your server instance operating system is not difficult with a little guidance to get started.

Our goals for this chapter include understanding the basic Ubuntu command line, updating and installing any operating system or packages required, installing both R and RStudio Server, and using R on both the command line and via the server. In particular, you'll learn how to get an R process running on the server regardless of whether it's connected to that server. Thus, we can use PuTTY to check in as needed (along with AWS warning signals) to detect when a process has finished.

Common Commands

Several of the commands we use in Ubuntu have counterparts in R. Others are more operating system specific rather than program specific. Many of the techniques we use to manage files in R can be done faster and more naturally via the command line. In this section, we explore the usual sorts of commands that are helpful to know and understand.

The first useful command is `clear`, a counterpart to Ctrl+L in R. This clears your viewing screen and allows you to easily see current commands and their results. We pair this command with `ls`, which is the list directory comments command. Right away the file we uploaded in the last chapter `ch02_link.txt` shows up.

```
clear
ls
ch02_link.txt
```

Now that we can see which files are in our directory, let's get a little bit more information about those files. The `ls` command can be modified with several letters. The `-l` modification is the long format, and it gives information about the file rights (`-rw-rw-r--`), about the user and the user group (both `ubuntu`, as we have a simple user set up currently), file size, as well as the date modified (notice this preserved our file's original modified date from before we uploaded it). Of particular note is the usage of file rights that relate to file security and safety. In Unix systems such as Ubuntu, all objects come with rights. The pattern is `-oooggpppp-`, where `o` refers to file owner, `g` relates to the group membership of the owner, and `p` is for public. Furthermore, each level of user can either Read, Write, or eXecute a particular object. This becomes quite important later, when we create some public-facing utilities that are accessible from the Web. It is important that we check the permissions, to ensure that the public can read files, yet not write or modify them. In our code that follows, both the user `ubuntu` as well as any future members of the group `ubuntu`

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_12

are allowed both read and write access to our text file. The general public, if this directory were ever made public, would be able to only read our file.

```
ls -l
total 4
-rw-rw-r-- 1 ubuntu ubuntu 14 Feb 14 05:16 ch02_link.txtCloud Ubuntu, Windows userscommands
```

We said the word *directory*, so we may as well see where we live in our system. Folders, or directories, also come with the same access permissions (they are objects), and we can create new directories and change directories. To present the working directory, we use `pwd;` to make a directory, we use `mkdir`; and to change the directory, we use `cd`. We show these commands and their results in the code that follows. Note that the `..` command returns us one level up in the directory structure, back to our original user directory.

```
pwd
/home/ubuntu

mkdir MyFolder
cd MyFolder
ubuntu@ip-10-0-0-244:~/MyFolder$ pwd
/home/ubuntu/MyFolder

ubuntu@ip-10-0-0-244:~/MyFolder$ cd ..
ubuntu@ip-10-0-0-244:~$
```

Having discussed making and navigating directories, we turn our attention to removing files. To remove a file, we use the command `rm`. We go ahead and do that to our text file that we uploaded; there was never any need for it other than as an example file. We show the removal in the following code, along with a full view of our user folder. Showing our list command with the full view allows us to see that our text file has been removed in the before and after views.

```
ls -hl
total 8.0K
-rw-rw-r-- 1 ubuntu ubuntu   14 Mar 16 02:29 ch02_link.txt
drwxrwxr-x 2 ubuntu ubuntu 4.0K Mar 16 14:09 MyFolder

rm ch02_link.txt
ls -hal
total 4.0K
drwxrwxr-x 2 ubuntu ubuntu 4.0K Mar 16 14:09 MyFolder
```

The values of these access permissions may be modified. The command `chmod` takes three number inputs after it in the pattern `chmod ogp`. We create a file and give three levels of access to the user, to the user's group, and to the public. Execute permission is given by `1`, write is provided by `2`, and read is given by `4`. Additionally, numbers may be added to give more permission. See the example that follows: `7 = 1 + 2 + 4` gives the user full access, and `6 = 2 + 4` gives the group read and write access only. Finally, the public has only write access.

```
touch permissionsFile
ls -hl
total 4.0K
drwxrwxr-x 2 ubuntu ubuntu 4.0K Mar 16 14:09 MyFolder
```

```
-rw-rw-r-- 1 ubuntu ubuntu    0 Mar 16 15:38 permissionsFile
chmod 762 permissionsFile
ls -hl
total 4.0K
drwxrwxr-x 2 ubuntu ubuntu 4.0K Mar 16 14:09 MyFolder
-rwxrw--w- 1 ubuntu ubuntu    0 Mar 16 15:38 permissionsFile
```

We end this section discussing two terminal commands, `sudo` and `whoami`. In the next section, we are going to start updating and maintaining our operating system. This requires root-level privileges. For security, our Ubuntu instance does not have a permanent root user, exactly. Instead, when we need elevated privileges, we give them to our `ubuntu` user by prefacing a command with `sudo`. This command stands for *superuser do* and is always followed by another command. The system temporarily escalates privileges for the command that follows.

```
whoami
ubuntu

sudo whoami
root
```

Now that we have the ability to run various commands as `root`, we turn to the next section, where we make sure that our cloud instance is up-to-date with the latest patches and packages.

Superuser and Security

Just as in any operating system and software suite, often updates should occur. Some may add new or improved functionality, while others may be more security related. While comprehensive cloud and server security is far beyond the scope of this text, a few steps make sense to do directly. Looking at Figure 12-1, it is clear we have some work to do.

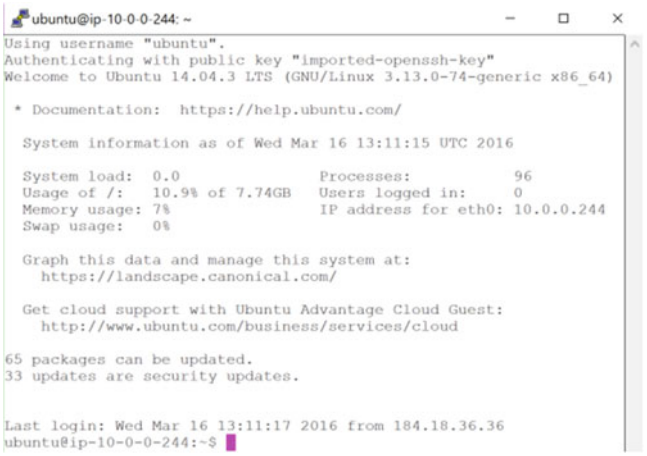


Figure 12-1. *The first screen shows 33 security updates*

The update process involves three steps. We first run `sudo apt-get update` with the following abbreviated code output:

```
sudo apt-get update
....
Fetched 3,451 kB in 3s (1,097 kB/s)
Reading package lists... Done
```

Next, we upgrade our current files so that we are ready to install new ones if we so desire. This takes some time, as we have quite a few to download! Again, we abbreviate some of the output. Of interest is to note that we did agree to continue when asked.

```
sudo apt-get upgrade
Reading package lists... Done
Building dependency tree
Reading state information... DoneCloud Ubuntu, Windows userssuperuser and security
Calculating upgrade... Done
....
62 upgraded, 0 newly installed, 0 to remove and 4 not upgraded.
Need to get 23.5 MB of archives.
After this operation, 209 kB of additional disk space will be used.
Do you want to continue? [Y/n] Y
....done.
```

Now that our files are upgraded, we can install our new packages. We run the command `sudo apt-get dist-upgrade` at the command line, agreeing to the prompts when asked. While it is not necessarily required, we also `reboot` the system. This exits us from our session, and we have to re-access our cloud via PuTTY. We show both the abbreviated code to perform these commands as well as the Figure 12-2 screenshot showing no more updates possible.

```
sudo apt-get dist-upgrade
...
sudo reboot
ubuntu@ip-10-0-0-244:~$
Broadcast message from ubuntu@ip-10-0-0-244
        (/dev/pts/0) at 16:40 ...

The system is going down for reboot NOW!
```

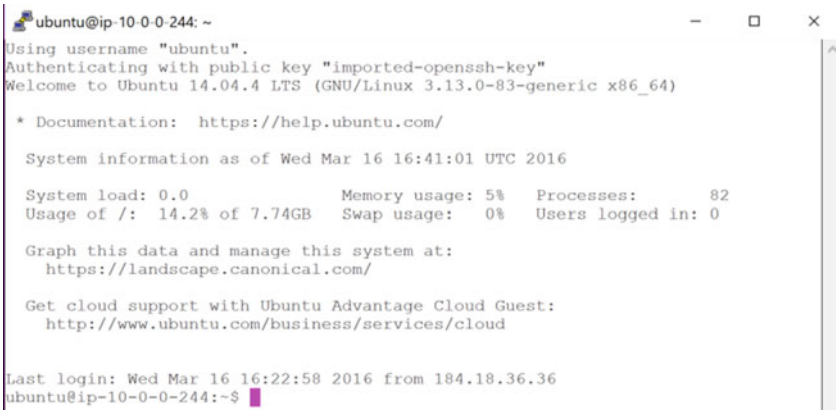


Figure 12-2. *Ubuntu startup screen showing no needed updates or upgrades*

Depending on when you read this, AWS may or may not have updated the version of Ubuntu from version 14.04 to 16.04. It is not required to upgrade. In fact, it is usually more convenient to wait until Amazon takes care of such things. However, should you wish to run on the latest version, run the command **`do-release-upgrade`**. Now, you are warned that upgrading via remote access is not advised and that additional access `ssh` scripts are being run to ports that are not likely open to the public. In particular, our security settings would prevent access to this backup access port. Since you have just created this instance, and can readily terminate from the AWS website, there is no need for such a backup access route. Hence, we simply ran the update, agreed to all the options when asked, and restarted.

If you do choose to upgrade the distribution, be sure to run the `sudo` update and upgrade the preceding commands one last time.

Now that our instance, or at least our packages on our instance, are updated, and we have a better sense of how to navigate in the Ubuntu file structure, we may turn our attention to installing R on our cloud server.

Installing and Using R

Our main goal so far in the last chapter and this one is to prepare a server to host R for us. We are very close to being there and have just a few more steps to perform. Our first step is to modify our package sources list so that Ubuntu knows where to find R:

```
cd /etc/apt/
ubuntu@ip-10-0-0-244:/etc/apt$ ls
apt.conf.d     sources.list    trusted.gpg
preferences.d  sources.list.d  trusted.gpg.d
sudo nano sources.list
```

Now that the `nano` text editor is open, arrow down to the end of the file and add the following line of code. Then press Ctrl+O to save; you see your filename and hit Enter/Return. From there, press Ctrl+X to exit to the command line. We should note that `trusty` and `xenial` are the version 14 and 16 names for Ubuntu, so in one go we ensure that either one will work.

```
#Add R File server
deb http://cran.rstudio.com/bin/linux/ubuntu trusty/

deb http://cran.rstudio.com/bin/linux/ubuntu xenial/
```

We should mention that we are using RStudio's kindly hosted mirror. Now, we move back into our home directory, and we also add the signature key E084DAB9 to our package utility. This allows us to confirm that any downloads we make are likely safe to download and install.

```
ubuntu@ip-10-0-0-244:/etc/apt$ cd /home/ubuntu
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv-keys E084DAB9

Executing: gpg --ignore-time-conflict --no-options --no-default-keyring --homedir /tmp/
tmp.jYANiNzPMT --no-auto-check-trustdb --trust-model always --keyring /etc/apt/trusted.gpg
--primary-keyring /etc/apt/trusted.gpg --keyserver keyserver.ubuntu.com --recv-keys E084DAB9
gpg: requesting key E084DAB9 from hkp server keyserver.ubuntu.com
gpg: key E084DAB9: public key "Michael Rutter <marutter@gmail.com>" imported
gpg: Total number processed: 1
gpg:               imported: 1  (RSA: 1)
```

From here, installing R is simple. In fact, it simply takes two commands. We have to update the packages list and then we need to install base R. In the setup process, you need to agree to some options once or twice.

```
sudo apt-get update
sudo apt-get install r-base
```

Now that we have installed R on our cloud, we can run R and see what happens. We use the command R to start our program. We run one of our scripts from Chapter 1 to see what happens. The following code shows both the start screen of R as well as the familiar results of running our code:

```
R

R version 3.3.1 (2016-06-21) -- "Bug in Your Hair"
Copyright (C) 2016 The R Foundation for Statistical Computing
Platform: x86_64-pc-linux-gnu (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

  Natural language support but running in an English locale

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

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

```
x <- c("a", "b", "c")
x[1]
[1] "a"
is.vector(x)
[1] TRUE
is.vector(x[1])
[1] TRUE
is.character(x[1])
[1] TRUE
```

Inside R, all the commands we regularly use work as expected. We can install packages or run code. However, we would like to improve three aspects. The first is that we are running R as a user, namely, the user `ubuntu`. It may well be that we wish other users of our server to be able to use R without needing to have `sudo` or `root` privileges. In that case, it may be helpful to install packages not just for our user, but for the entire machine. Second, running code from files that we have created on our local machines would be convenient. Third, our connection to our server lives at the mercy of the Internet. If there is a glitch, we would lose our work. Thus, it is also convenient to get to the point where we can set our R commands to run from the command line so that even if we disconnect, our server is still calculating away.

To install packages for just a particular user, you use the `install.packages("")` function inside the R console as usual. However, to install a package for any user (including future users you may invite to your cloud), use the following command. What you need to do is pass a command from the command line to R. We also need to run this as `root`, and we need root privileges to do so, which `sudo` and `su -` provide. The `-c` option executes the code directly following it. Finally, we call R and set it to install the package.

```
sudo su - -c "R -e \"install.packages('pscore', repos = 'http://cran.rstudio.com/')\""
```

To run code already created, we use the command `R CMD BATCH`. It can help to write the commands we want to use in RStudio on our local computer, use `WinSCP` to upload that `*.R` file to our instance, and then run the file as follows:

```
R CMD BATCH chapter21.R
ls
chapter21.R  chapter21.Rout  MyFolder  permissionsFile
```

While the file `chapter21.R` may be downloaded from our code repository with Apress, it is simply the snippet of loop code from a preceding chapter that cubes several numbers. We can see the output in the file `chapter21.Rout` and we can view it by using the `nano` editor we used earlier in this chapter. Remember that Ctrl+X exits from that editor. We show both the command to open the file as well as an abbreviated output to demonstrate that the code ran successfully:

```
nano chapter21.Rout
```

```
....
> head(xCube)
[1]   1   8  27  64 125 216
> forTime
   user  system elapsed
 11.468   1.013  12.487
....
```

We may now run R commands on our instance at will. This includes the parallel-processing techniques you learned in earlier chapters. Provided our code is built not to require user input, it may be readily left to run, provided you have a solid connection. We do this with the `screen` command. Essentially, this opens another terminal from which we can detach. This allows that terminal to keep running even if our PuTTY terminal session ends (either because we wish it to end or because network vagrancies force our hand). We can see when our process is done running as well. The command we run after our `screen` command with modifiers is familiar; should you wish more details about `screen`, the command `man screen` gives plenty of information.

```
screen -A -d -m R CMD BATCH chapter21.R
screen -ls
There is a screen on:
        1525..ip-10-0-0-244     (03/17/2016 04:43:40 AM)        (Detached)
1 Socket in /var/run/screen/S-ubuntu.

screen -ls
No Sockets found in /var/run/screen/S-ubuntu.
```

As you can see from the second listing, no more processes are running. Thus it is safe to access our output file to see the results. For a process that took longer to run, rather than enter `screen -ls`, we might type `exit` and go rest. For processes that run on pricier instances, it is possible to set up AWS to e-mail an alert when processor use drops below a certain threshold.

We now have installed R on our server instance, can set up packages that are accessible to all users, and can run files without maintaining a steady network connection. If we design code on one machine that seems to take too long to run, we can upload those files to a more powerful machine and run it. However, we are still using the command line. In the next section, we install RStudio on our instance, so that we can use the familiar look and feel of RStudio while still gaining the power of a cloud instance.

Installing and Using RStudio Server

While we have shown that it is relatively straightforward to upload a file that was running too slowly on a local machine to our cloud instance, the format does leave something to be desired. `RStudio` (RStudio Team, 2015) is a wonderful, intuitive environment in which to code R, and we can get that same graphic interface. What is more, this can be accessible from a browser rather than from the command line. This can be quite convenient, as just about any Internet-capable device with a browser can access your instance and use R.

The risk, of course, is some security concerns. RStudio Server is protected by only a password rather than a key file. Recall from Chapter 2 how many system administrative tasks, such as file moving and deletion, may be done by R. Access to RStudio Server may enable a dedicated attacker more access to your system than desirable. We, of course, continue to keep our security group on AWS set up to allow connections only from our local IP address, although this rather curtails the promise of use of R from any device. We remind you again that data security is worth a closer look than we give here. We turn to our cloud and installation.

Since we have already installed R, we need only a little bit of work after accessing our command line again via PuTTY. To ensure proper package authentication, it helps to update, just as when we were installing system and security updates. As a side note, while installations often require a yes or no input, if asked to install packages without verification, there is likely a better way that involves verification. After each of the following `apt-get` commands, much text prints to your console, and you may be prompted to agree to some disk space usage.

```
sudo apt-get update
...
sudo apt-get install gdebi-core
...
```

Once you have the ability to install the server, you need to download it from the Web. Run the following command, and expect to download a file about 51 M in size. We show the code and some of the output here; be sure to note the saved name of the file as well as a full download being complete:

```
wget https://download2.rstudio.org/rstudio-server-0.99.903-amd64.deb
...
2016-10-02 15:09:44 (10.5 MB/s) - 'rstudio-server-0.99.903-amd64.deb' saved
[53985492/53985492]
```

Now we install the server. This should require one yes agreement during the installation process:

```
sudo gdebi rstudio-server-0.99.903-amd64.deb
Reading package lists... Done
Building dependency tree
Reading state information... Done
Building data structures... Done
Building data structures... Done
...
Do you want to install the software package? [y/N]:y
Selecting previously unselected package rstudio-server.
(Reading database ... 86981 files and directories currently installed.)
Preparing to unpack rstudio-server-0.99.892-amd64.deb ...
Unpacking rstudio-server (0.99.892) ...
Setting up rstudio-server (0.99.892) ...
groupadd: group 'rstudio-server' already exists
rsession: no process found
rstudio-server start/running, process 2445
```

To confirm that the server is running, run the verification commands as follows:

```
sudo rstudio-server verify-installation
rstudio-server stop/waiting
rstudio-server start/running, process 2605Cloud Ubuntu, Windows usersRStudio Server
```

Now that we know our server is running, we want more confirmation that it is working. Remember in the last chapter that we set our security group to allow access to RStudio Server on port 8787. Open a web browser to your server's web address at the needed port, namely, `54.201.117.64:8787` (or for you, `YOUR_INSTANCE_IP_HERE:8787`). As you can see in Figure 12-3, it does indeed visually appear that the server is working.

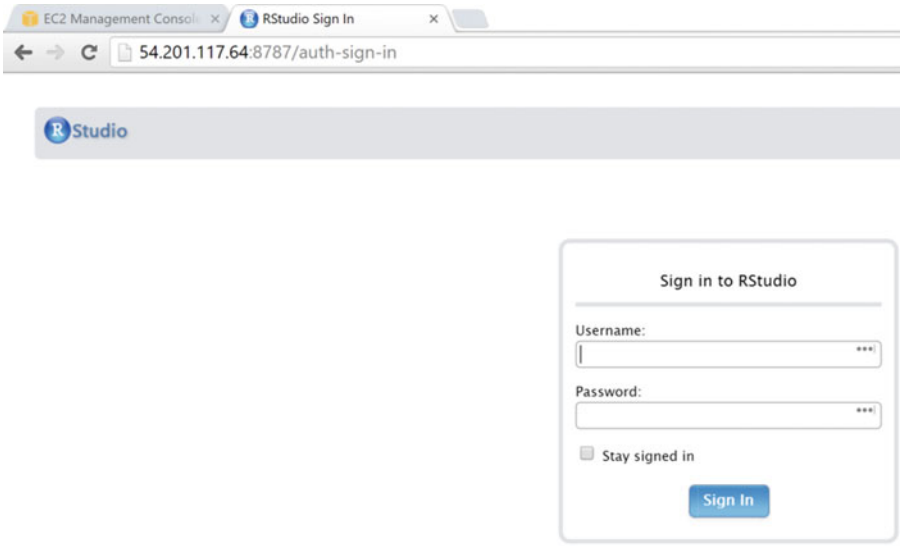


Figure 12-3. *RStudio Server*

We need a username and password to gain further access. The user is also a user of your Ubuntu server and has a password. It is wise to have a password logon to the server disabled by default, which our instance does in fact have. The command we use to create a new user is `adduser`, and this runs the script shown for us. We need to type these commands back in our command line. It is not necessary to fill in information for your new user beyond the username and password. You can simply press Enter/Return to move to the next line. Also, note that the password field has suppressed keystrokes so that nothing populates there.

```
sudo adduser advancedr
Adding user `advancedr' ...
Adding new group `advancedr' (1001) ...
Adding new user `advancedr' (1001) with group `advancedr' ...
Creating home directory `/home/advancedr' ...
Copying files from `/etc/skel' ...
Enter new UNIX password:
Retype new UNIX password:
passwd: password updated successfully
Changing the user information for advancedr
Enter the new value, or press ENTER for the default
        Full Name []: Advanced R
        Room Number []: RStudio Server
        Work Phone []:
        Home Phone []:
        Other []:
Is the information correct? [Y/n] Y
```

Going back to our browser, we use our new username and password to access RStudio Server. Enter the username and password just created, as in Figure 12-4, and you should be into RStudio on the Web.

Figure 12-4. *Sign into RStudio*

This should feel just like the RStudio we have been using all along, as shown in Figure 12-5.

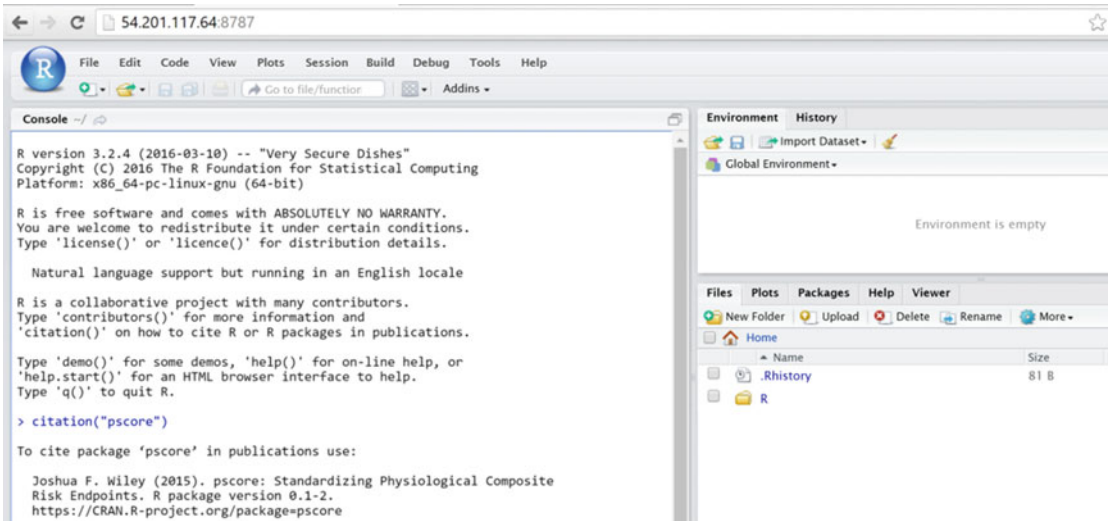


Figure 12-5. *RStudio on your cloud*

RStudio now exists on your cloud instance. While this is helpful for a more intuitive way to work with R, the ability to more natively use and see graphical output is something we, the authors, would like to stress. Such output may be readily saved to `*.pdf` files and downloaded via `WinSCP` from the `/home/advancedr` folder. More generally, `/home/YOUR_USERNAME` accesses a particular RStudio user folder.

This completes our setup of RStudio Server. More could be done, but this is enough for starters. You now have access to a potentially powerful system, and indeed are free to have several duplicates of such systems if you wish. On AWS, it is possible to set up warnings that alert you via e-mail when your instance(s) drop below a certain CPU utilization rate. Thus, an R process may be configured to run, and instance space need not be paid for much beyond actual use time. Our final suggestion is to spend some time with the RStudio Server documentation if you intend to use it more than occasionally. There are some useful customizations, convenient features, and good-to-know caveats. We include the link in the reference section of this chapter.

Installing Microsoft R

Our cloud instance is `t2.micro`. As such, it has only one virtual CPU and gains no benefit from parallel-processing techniques. We have also already installed base R, so R is of course working. However, on the chance you choose to spin up more-powerful instances, we also install Microsoft R Open (MRO) and `MKL` (Microsoft, 2016). These files need to be installed after downloading them. We are going to give two options here. First, we install the latest version of Microsoft R Open, which is version 3.3.1 as of this writing. We also show how to install an earlier version, should you happen to have code you intend to run that requires older versions of R.

The main difference is that in the past, MRO and MKL were two separate installations. Now, starting with version 3.3.1, it is a single installation process. Type the following command in bold; the output follows:

```
wget "https://mran.microsoft.com/install/mro/3.3.1/microsoft-r-open-3.3.1.tar.gz"
--2016-10-02 16:52:02--  https://mran.microsoft.com/install/mro/3.3.1/microsoft-r-open-
3.3.1.tar.gz
Resolving mran.microsoft.com (mran.microsoft.com)... 166.78.134.173, 2001:4800:7813:516:be7
6:4eff:fe04:4c9b
Connecting to mran.microsoft.com (mran.microsoft.com)|166.78.134.173|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 268235803 (256M) [application/octet-stream]
Saving to: 'microsoft-r-open-3.3.1.tar.gz'

microsoft-r-open-3.3.1.tar.gz 100%[=============================================>]
255.81M  18.8MB/s    in 13s

2016-10-02 16:52:15 (20.1 MB/s) - 'microsoft-r-open-3.3.1.tar.gz' saved
[268235803/268235803]
```

Once the file has been downloaded, we must decompress our downloaded file. This requires privileges, so we use `sudo`. Using `tar`, we extract and ungzip our file:

```
sudo tar -xzf microsoft-r-open-3.3.1.tar.gz
```

Next, we change our directory to our newly unzipped files by using the change directory command:

```
cd microsoft-r-open
~/microsoft-r-open$ ls
deb  install.sh  MKL_EULA.txt  MRO_EULA.txt  rpm
```

When installing, be sure to carefully note when to type `q` and when to type `y`. Again, the installation using the `install.sh` file will require `sudo` privileges:

```
~/microsoft-r-open$ sudo ./install.sh

Press [Enter] key to display the Microsoft R Open license. When finished reading, press q to
continue:

Do you wish to install the Intel MKL libraries?
Choose [y]es|[n]o y

Press [Enter] key to display the Intel MKL license. When finished reading, press q to
continue:
```

```
Do you agree to the terms of the previously displayed license?
Choose [y]es|[n]o y

Updating apt package repositories...done
Installing apt package dependencies libxt6 libsm6 libpango1.0-0 libgomp1 curl...done
Installing /home/ubuntu/microsoft-r-open/deb/microsoft-r-open-mro-3.3.deb...done
Installing /home/ubuntu/microsoft-r-open/deb/microsoft-r-open-foreachiterators-3.3.deb...
done
Installing /home/ubuntu/microsoft-r-open/deb/microsoft-r-open-mkl-3.3.deb...done

Thank you for installing Microsoft R Open.
You will find logs for this installation in
/home/ubuntu/microsoft-r-open/logs
```

This completes the installation, and if you were on an instance that allowed for multiple threads, you would have access. This bundle of MRO and MKL is new, and thus, for completeness for backward compatibility, we include instructions for the older file type as well. To install older versions such as 3.2.4, in the command line, use the `wget` command:

```
wget "https://mran.microsoft.com/install/mro/3.2.4/MRO-3.2.4-Ubuntu-14.4.x86_64.deb"
```

Then type the following:

```
wget "https://mran.microsoft.com/install/mro/3.2.4/RevoMath-3.2.4.tar.gz"
```

After those have downloaded, use the `ls` command to see that it worked:

```
ls
chapter21.R                      permissionsFile  RevoMath-3.2.4.tar.gz
slider_time
MRO-3.2.4-Ubuntu-14.4.x86_64.deb  PieChart_Time    rstudio-server-0.99.892-amd64.deb
Upload_hist
MyFolder                         R                shiny-server-1.4.2.786-amd64.deb
```

Next, enter the following code into the command line:

```
sudo dpkg -i MRO-3.2.4-Ubuntu-14.4.x86_64.deb
```

Now that Microsoft R Open is installed, we add the parallel-processing support:

```
sudo tar -xzf RevoMath-3.2.4.tar.gz
cd RevoMath
ubuntu@ip-10-0-0-244:~/RevoMath$ ls
mkl  mklLicense.txt  RevoMath.sh  RevoUtilsMath.tar.gz
ubuntu@ip-10-0-0-244:~/RevoMath$ sudo ./RevoMath.sh
```

After that last line, you want to select option 1 by typing `1` and then pressing Enter/Return. Hit Enter/Return a few times to get through the license agreement text (depending on screen size), but not too many times or you negate the agreement. When prompted, type `y` and hit Enter/Return. That's it! Should you have more than one core, the cloud now works as MRO has been working for us on Windows.

Installing Java

Next, we install Java. It is quite easy; just run the following three lines of code, one at a time, in the command line. Agree to things as needed.

```
sudo apt-get update
sudo apt-get install default-jre
sudo apt-get install default-jdk
```

Installing Shiny on Your Cloud

Bringing statistics to information consumers is the main use for Shiny. Because we are in installation mode, we'll go ahead and install the needed resources on our cloud instance to host a Shiny application.

To install Shiny, we need to run the following code. An agreement is required to confirm an installation, and we suppress the system's output to the command line. Run each line by itself.

This line installs the `shiny` package (Chang, Cheng, Allaire, Xie, and McPherson, 2016) into R. In addition, we also install `xlsx` (Dragulescu, 2014) and `shinydashboard` (Chang, 2015). You notice that between the inner parentheses, there are commands that make lots of sense to R users. However, in this case, we are running R from the command line, specifically, so we can execute the process with administrative privileges and thus have access to these libraries from any Ubuntu user account rather than just the main user account. When a Shiny server hosts applications, it hosts them by using the username `shiny`. Hence, the following three lines of code install these packages for all users:

```
sudo su - -c "R -e \"install.packages('shiny', repos='https://cran.rstudio.com/')\""

sudo su - -c "R -e \"install.packages('xlsx', repos='https://cran.rstudio.com/')\""

sudo su - -c "R -e \"install.packages('shinydashboard', repos='https://cran.rstudio.
com/')\""
```

From here, now that R itself can use Shiny, we install the server so that our cloud instance can serve Shiny apps to the Internet. We have to download a file with the `wget` command and then install that file. We do that with the following lines of code, and there are some installation disclaimers.

```
wget https://download3.rstudio.org/ubuntu-12.04/x86_64/shiny-server-1.4.6.809-amd64.deb

sudo gdebi shiny-server-1.4.6.809-amd64.deb
```

Final Thoughts

You now have access to a cloud instance that has RStudio Server (which can be quite convenient). In particular, you have the ability to upload files to and from a local machine in a way that does not require superuser local privileges. If you opened up permission access by unrestricting more IP addresses, this is a way to use R from a smartphone. Also, we installed several packages and applications on our Ubuntu server. In the next chapters, we use those features to make some absorbing analytics happen. More important, because they are served on the cloud, these data and results are accessible to anyone. Thus, sharing results has never been easier.

CHAPTER 13

Every Cloud has a Shiny Lining

When serving data to the public, easy access and interactive information make a real difference in comprehension. In this chapter, our goal is to provide access and some interesting uses for a more recent application, `shiny` (Chang, Cheng, Allaire, Xie, and McPherson, 2016).

Shiny is a web application framework for R. What it does is allow information consumers to interact with live data and analytics in R . This framework does not require knowledge of any web-based languages. All the same, surprisingly complex and interactive data models can be released to your clients, decision makers, or consumers.

Do you have data your customers should explore? Are there performance metrics that, transformed from static to interactive, might support your story to a board or a boss? Do your users have data they need to upload and understand better? Would you like to easily update next quarter's data into a common dashboard? All these questions can be answered with Shiny—and since Shiny is served via a web page, if we can get Shiny onto our cloud, then anyone can benefit from the power of R.

Now, for simplicity's sake, in this chapter we are going to live entirely on Windows inside RStudio. In the previous chapter, we got Shiny Server running on our cloud, but for now, you'll learn on your local machine. In the next chapter, we put all the building blocks we had in this chapter into one dashboard, and we upload that to our cloud.

The Basics of Shiny

Think of Shiny as a web interface for R code. If you can do it in R, Shiny can push those results to a nice page that is viewable via Internet browsers. Even more awesomely, if a data consumer wants to interact with information, their input can be incorporated into R code.

In `RStudio` (RStudio Team, 2015), select File ➤ New File ➤ Shiny Web App. Name your file `PieChart_Time`, and under Application choose the single file option (`app.R`). Delete all code and type in the following code (or download from the Apress site). As you type it in, you want to notice three things. First, there is a user-interface portion of the code. Second, there is a server side to the code. Finally, the application must be run. Another aspect to notice (and this is why we recommend typing it all in manually) is where pieces of the user side and the server side mesh. In fact, in our code, there are exactly four places this happens. From the user side, variables of a number between `0` and `100`, a text string, and a check box are input. The server receives those values and creates plots. It puts those plots into the output. The user side gets that output in `mainPanel()`.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_13

```
library(shiny)

# user interface drawing a pie chartShinymainPanel()
ui <- shinyUI(fluidPage(

   # Page title
   titlePanel("User Controlled Chart"),

   # Sidebar with numeric, text, and check box inputs controlled by user
   sidebarLayout(
      sidebarPanel(
         numericInput("pie",
                      "Percent of Pie Chart",
                      min = 0,
                      max = 100,
                      value = 50),

         textInput("pietext", "Text Input", value = "Default Title",
                   placeholder = "Enter Your Title Here"),

         checkboxInput("pieChoice",
                       "  I want a Pie Chart instead.", value = FALSE)
      ),

      # Show the plot(s)
      mainPanel(
         plotOutput("piePlot")
      )
   )
))

# server side R code creating Pie Chart or barplot hidden from user
server <- shinyServer(function(input, output) {

   output$piePlot <- renderPlot({
      # generate Pie chart ratios based on input$pie from user
      y <- c(input$pie, 100-input$pie)

      # draw the pie chart or barplot with the specified ratio and label

    if(input$pieChoice == FALSE){
      barplot(y, ylim = c(0,100),
              names.arg = c(input$pietext, paste0("Complement of ", input$pietext)))

    }else{
      pie(y, labels = c(input$pietext, paste0("Complement of ", input$pietext)))}

   })
})
```

```
# Run the application
shinyApp(ui = ui, server = server)
```

Before we delve into that code, let's see it in action. Type and run the code that follows from the RStudio console to install Shiny on your local Windows machine:

```
install.packages("shiny")
```

Now, go into your `app.R` file you carefully typed and run that code in one block. There should even be a Run App play button near the top right of the code area. Either way, something along the lines of Figure 13-1 should appear.

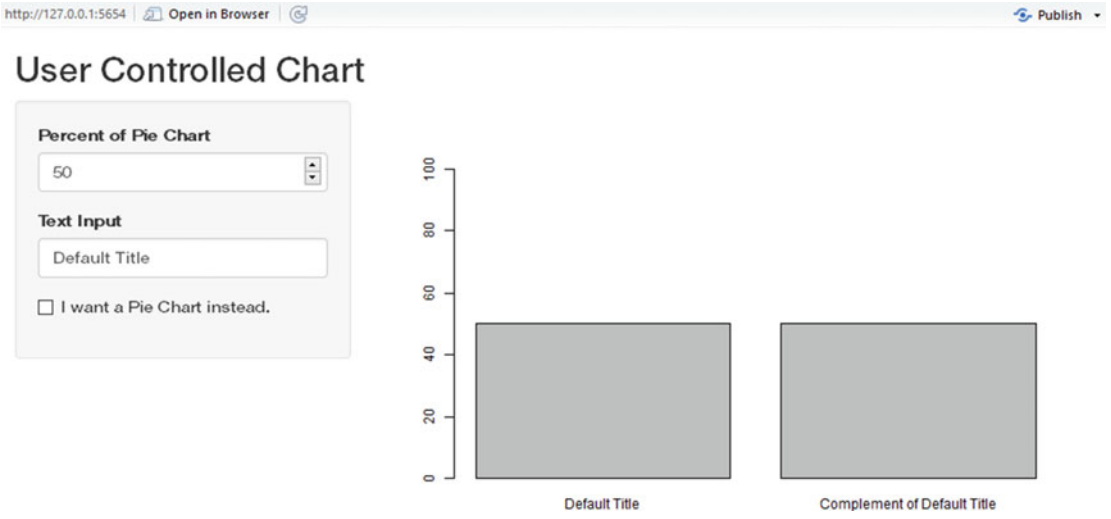


Figure 13-1. *A locally hosted Shiny App to see just what we are gaining*

Go ahead and enter various numbers and text values. Play with Shiny and see what it can do. While you do so, recognize that what Shiny has done is provide an interface between you and R. Now, this particular run is hosted by RStudio on your local machine. Notice that in the console it indicates listening, and you may need to select a Stop button before you can enter any new code.

So what happened? Suppose you changed the Percent of Pie Chart number to **75**. Well, that `numericInput()` is a reactive value. When you change it, behind the scenes, a signal is sent alerting the server that `input$pie` has changed. The server then notifies any reactive functions that use `input$pie` that something has changed. In our case, `renderPlot()` is a reactive function. It now knows `input$pie` is different, so it queries that value, gets the new value, and runs all its code again. Since that is an output value, this whole cycle repeats in reverse with `output$piePlot`, with the result that `plotOutput()` on the user side refreshes.

There are some important things to notice about all this. The process we just described runs all the code again inside `renderPlot()`—all of it. Now, ceteris paribus, there is no need to look up `input$pietext` and `input$pieChoice`. Our reactive function knows its values, for those are up-to-date, and while they are used again, they are not called again. Still, if we had something a little more intensive than a subtraction and some plots inside `renderPlot()`, we might find ourselves with a relatively steep performance cost to making any changes. While this set of code is quite compact, nevertheless you can see three places where you might put R code. One is where `library(shiny)` is. This is on the outside of our `shinyServer()`. Code that is here,

such as our library call, is run once each session. A single session can have more than one user interacting with it (although that gets a bit esoteric at the moment). Still, each new user executes all the code inside the `shinyServer()` at least once. Finally, as we just saw, the code inside reactive functions such as `renderPlot()` are run once each update from any connected reactive input values. What this means is that for efficiency, consider placing as much code as possible outside the reactive functions. Moreover, if you anticipate heavy user traffic, place as much outside the server as possible too.

For now, let's introduce some new functions. Remember, there is the user input side, there is the server side, and there is the entire Shiny app. Deep down inside, Shiny is just those three pieces. In fact, here is a rather bare-bones Shiny application:

```
library(shiny)

# Define user interface
ui <- shinyUI(fluidPage())

# Define server
server <- shinyServer(function(input, output) {})

# Run the application
shinyApp(ui = ui, server = server)
```

You can run this code and get a nice blank page. While that is perhaps less inspiring than the previous code, this helps you get coding in this interface quickly. In the user-interface section, what we see is the `fluidPage()` function call, inside of which we place all our user-facing text. This area of the code is the most unfamiliar part of Shiny, because this is the code that creates the HTML seen by users. In the reference section, we place links to all the Shiny UI functions. However, for this chapter, we keep to the same basic layout inside a page that is fairly self-adjusting, or fluid, depending on the browser viewing area. Notice this is an R function, and functions in R take arguments that are separated by commas. Every aspect of our growing application is contained inside this wrapper function, which takes care of all the conversion to web-ready markup. For our first app from Figure 13-1, we used a sidebar layout. This function, in turn, wants two arguments: the sidebar panel where we control our application and then the main panel where we draw it. Editing our user interface gives us the following code:

```
# Define user interface
ui <- shinyUI(fluidPage(
  sidebarLayout(
    sidebarPanel(),
    mainPanel()
  )
  ))
```

This code is a wonderful example of the power of the mathematical notion of function composition. Our goal is to eventually have something the user inputs to our R code, such as the text or check box of our example in Figure 13-1. The user inputs those values into as-yet-to-be-named functions, which give visual output, which is input into the `sidebarPanel()` function, which provides web output, which is input to `fluidPage()`, which outputs the final web user interface. We can see the results, blank though they be, in Figure 13-2.

Figure 13-2. *A mostly blank Shiny app, with a sidebarPanel()*

From this point, we can quickly build our user's inputs with three input functions. Each of these input functions follows a specific format, namely `blargInput(input identification, text label for user readability, settings relating to the specific type of input for which blarg is a stand-in)`. Formally, our three input functions have these layouts, and please notice the similarities!

```
numericInput(inputId, label, value, min = NA, max = NA, step = NA,  width = NULL)
textInput(inputId, label, value = "", width = NULL, placeholder = NULL)
checkboxInput(inputId, label, value = FALSE, width = NULL)
```

Proceeding to fill in some specifics for our code, we can set the input identities for each to be useful variables for us to use later in our server code. Since we do plan to build a pie chart (and please, gentle readers, do not hate us for that choice), it makes sense to limit our numeric input to between `0` and `100`. Because our server code runs once on start, it also makes sense to provide a default value for the numeric input and, indeed, the text input. Building further on our burgeoning user-interface code, we now have this:

```
# user interface drawing a pie chart
ui <- shinyUI(fluidPage(
  # Sidebar with numeric, text, and check box inputs controlled by user
  sidebarLayout(
    sidebarPanel(
      numericInput("pie", "Percent of Pie Chart",
                   min = 0, max = 100, value = 50),
      textInput("pietext", "Text Input", value = "Default Title",
                placeholder = "Enter Your Title Here"),

      checkboxInput("pieChoice",  "  I want a Pie Chart instead.", value = FALSE)
              ),

    mainPanel()
)
))
```

Again, after making those modifications, it makes sense to run the whole app again, and we see the new result in Figure 13-3. Notice that the `mainPanel()` is currently empty in our code, and, indeed, in Figure 13-3 we do not see anything on the right-hand side.

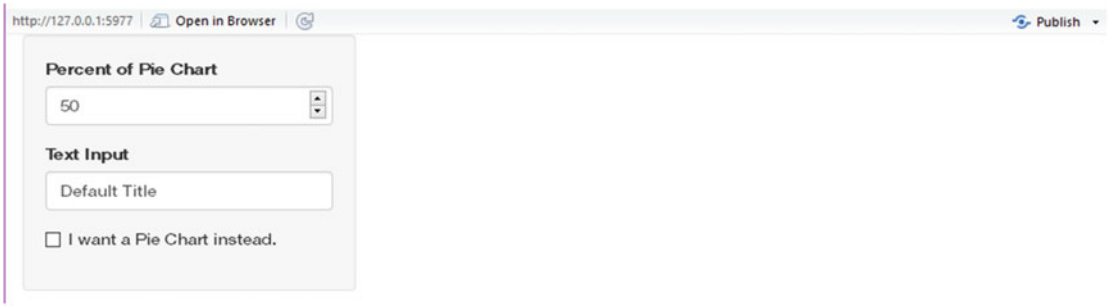


Figure 13-3. *The user interface is almost done, and the input portion in the sidebar is indeed done*

Now that we have our user input captured, we are ready to use it on the server side. Remember, Shiny is an R wrapper that takes input, and when that input changes, alerts a reactive function on the server side that something is different. Behind the scenes, Shiny is taking care of knowing when the input changes, and alerting the correct reactive functions on the server; we need not worry about such things. Our server defines a new function that takes both input and output as formals, and uses those in its body. If needed, take a quick look back at Chapter 4 to refresh your memory on just what a function is. Since all we want to do is draw either a bar plot or a pie chart, the majority of our code does precisely that in very common R code. In fact, the only part that is not fairly normal R code is our reactive function `renderPlot()`. Since we wish to give graphical outputs, we use that function.

Reactive functions are always waiting for Shiny to alert them to any changes in the inputs from the user-interface side. When an input changes, the reactive functions are notified if they have at least one of the changed values in their body. If they do, then the reactive functions run their code again, this time with the new value(s). We see this in the following code. The only other part of this is that the output of `renderPlot()` is stored in an output variable, and we choose that name to be `piePlot` (even though it might not output a pie chart, depending on the check box choice—which is wholly our fault for selecting a poor name).

```
# server side R code creating Pie Chart or Bar plot
server <- shinyServer(function(input, output) {

   output$piePlot <- renderPlot({
      # generate Pie chart ratios based on input$pie from user
      y <- c(input$pie, 100-input$pie)

      # draw the pie chart or barplot with the specified ratio and label

    if(input$pieChoice == FALSE){
      barplot(y, ylim = c(0,100),
              names.arg = c(input$pietext, paste0("Complement of ", input$pietext)))

    }else{
      pie(y, labels = c(input$pietext, paste0("Complement of ", input$pietext)))}

   })
})
```

That is essentially it. When our server runs, it alerts the user-interface function `plotOutput("piePlot")` to the new image. This function lives in the `mainPanel()` region. Under this simple model of Shiny, the

sidebar is where user input is received, the server side is where it is processed behind the scenes by R, and the main panel is where the processed output is viewed by the user. Just this once, we show the full HTML code that Shiny creates for us to serve in Figure 13-1:

```
<html>
<head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>  <script
type="application/shiny-singletons"></script>  <script type="application/html-dependencies">
json2[2014.02.04];jquery[1.11.3];shiny[0.13.1];bootstrap[3.3.5]</script><script src="shared/
json2-min.js"></script>
<script src="shared/jquery.min.js"></script>
<link href="shared/shiny.css" rel="stylesheet" />
<script src="shared/shiny.min.js"></script>
<meta name="viewport" content="width=device-width, initial-scale=1" />
<link href="shared/bootstrap/css/bootstrap.min.css" rel="stylesheet" />
<script src="shared/bootstrap/js/bootstrap.min.js"></script>
<script src="shared/bootstrap/shim/html5shiv.min.js"></script>
<script src="shared/bootstrap/shim/respond.min.js"></script>  <title>User Controlled Chart
</title>
</head>
<body>
  <div class="container-fluid">
    <h2>User Controlled Chart</h2>ShinyHTML code, creation
    <div class="row">
      <div class="col-sm-4">
        <form class="well">
          <div class="form-group shiny-input-container">
            <label for="pie">Percent of Pie Chart</label>
            <input id="pie" type="number" class="form-control" value="50" min="0"
             max="100"/>
          </div>
          <div class="form-group shiny-input-container">
            <label for="pietext">Text Input</label>
            <input id="pietext" type="text" class="form-control" value="Default Title"
placeholder="Enter Your Title Here"/>
          </div>
          <div class="form-group shiny-input-container">
            <div class="checkbox">
              <label>
                <input id="pieChoice" type="checkbox"/>
                <span>  I want a Pie Chart instead.</span>
              </label>
            </div>
          </div>
        </form>
      </div>
      <div class="col-sm-8">
        <div id="piePlot" class="shiny-plot-output" style="width: 100% ; height: 400px">
        </div>
      </div>
    </div>
```

```
  </div>
</body>
</html>
```

So now, go back and carefully look through the very first block of code we showed in this section that was the entire Shiny app and realize that code creates all of the preceding HTML and JavaScript. That is all there is to Shiny. Now, in this example, our R code was not particularly impressive. All the same, it is the principle that counts. Users can now interact with data, and it is not impossible to imagine more-interesting applications.

In the next few sections, we introduce a handful of other interesting and useful Shiny input functions along with some essential reactive functions. Finally, in the end, we upload all of them to our cloud so that you can see them live on the Web.

Shiny in Motion

Now that we have a better appreciation for what Shiny does, it is important to appreciate the dynamic potential of a web page over print reports. One of the authors is a mathematics professor, and one of the challenges in teaching mathematics is that once data goes beyond three dimensions, it becomes difficult for students to conceptualize that information. Naturally, students, being people, are not unique in their high-dimensional blindness. Time can provide a fourth dimension, and Shiny can provide us that time.

Another feature of Shiny we mentioned earlier, is that R code makes sense to place in three distinct locations. The first is at the beginning of all the code, before user-interface or server-side code is written. Code placed in this area runs once per instance. While the value of this may not be as clear on a local machine, when your code is hosted on a cloud, a particular R session may have more than one user interacting with it. That is the way the Shiny server we installed in the preceding chapter works. This is a great place to read in any static data or perform any once-only calculations. It is the most computationally efficient. While we do not do so in this example, the code may also be placed inside the server code but outside any reactive functions that would be run once per user connection to a session. Finally, the code inside reactive functions, inside the server function, is executed once every time any input data that the function references is changed.

Let's take a look at how to change data with respect to time. From the user-interface side, there is an input function that is a number-line slider bar. As with all input functions, we need to give it a name so that our reactive functions can know where to look for input updates. As with our `numericInput()` from the prior section, we give this function some minimum and maximum values in years. Finally, we are going to animate our `sliderInput()` function. Take a look at the code we use inside our user interface. Our code uses an animate option that includes an `interval` in milliseconds and also has the animation keep on looping.

```
sliderInput("ayear", "Academic Year:",
                     min = 2014, max = 2017,
                     value = 2014, step = 1,
                     animate = animationOptions(interval=600, loop=T) )
```

While this application also shows how to perform R calculations on both static data and user inputs, it really is quite simple again from the R code perspective. In fact, other than the `titlePanel()` command and of course our slider code, this is quite a bit like our first Shiny application. We show both the code and the end result in Figure 13-4.

```
library(shiny)
slider_data<-read.csv("slider_data.csv", header = TRUE, sep = ",")
Phase1    <- slider_data[,2]
Phase2    <- slider_data[,3]
Phase3    <- slider_data[,4]
Phase4    <- slider_data[,5]
```

```
# Define UI for application that draws Bar Plot
ui <- shinyUI(fluidPage(

   # Application title
   titlePanel("Training Programme Results"),

   # Sidebar with a slider input for number of bins
   sidebarLayout(
      sidebarPanel(
sliderInput("ayear", "Academic Year:",
                    min = 2014, max = 2017,
                    value = 2014, step = 1,
                    animate = animationOptions(interval=600, loop=T) )  ),

      # Show the bar plot
      mainPanel(plotOutput("barPlot") )               )               ))

# Define server logic required to draw a barplot
server <- shinyServer(function(input, output) {

   output$barPlot <- renderPlot({
      # Count values in each phase that match the correct date.
     cap<-input$ayear*100
     x <- c(sum(Phase1<cap), sum(Phase2<cap), sum(Phase3<cap), sum(Phase4<cap))

      # draw the barplot for the correct year.
      barplot(x, names.arg = c("Phase I", "Phase II", "Phase III", "Fellows"),
              col = c("deeppink1","deeppink2","deeppink3","deeppink4"),  ylim=c(0,50))
   })           })

# Run the application
shinyApp(ui = ui, server = server)
```

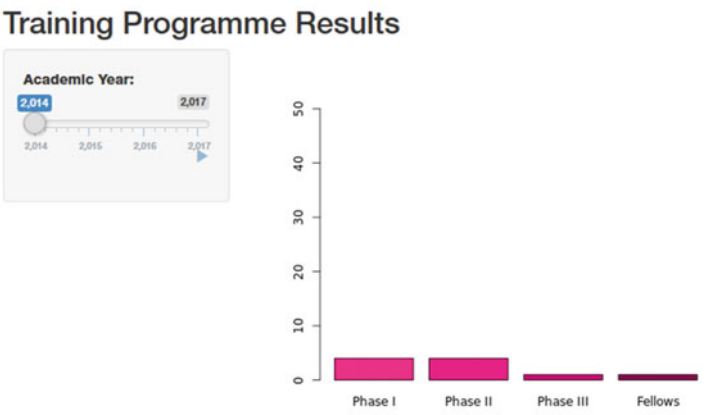


Figure 13-4. *A slider bar input that has a Play button feature showing data in motion*

Uploading a User File into Shiny

Our last example for this chapter indicates that users can upload files via Shiny that can then be processed via R. Of course, unlike our first two examples, this is a shade riskier because such uploaded files are not under our control. The data might be any file and not truly work with our results. It is possible to build various error-checking features into code; in fact, we mentioned such techniques in part while discussing functions. However, to focus our code on strictly new features, we pretend for a moment that we live in a perfect world. We have a sample file on the Apress website for this book that is just a comma-separated values file with the numbers 1 through 10.

Perhaps unsurprisingly, to receive a file input, the command in Shiny is `fileInput()`. Other than `inputId`, maybe the most interesting feature is that this could even accept `multiple` uploads, although in our example we leave that to its default value of `FALSE`. Uploading more than one file might not be supported in all browsers. Also in the user-interface area, we use a new output function, `tableOutput()`, which along with `plotOutput()` allows us to have both chart and graphical outputs for our users. Before we talk about the server-side code that enables us to interact with our user's file, we take a look at what our CSV file upload gives us. Figure 13-5 is the view after we uploaded our CSV file, and we show the code for just the user interface:

```
ui <- shinyUI(fluidPage(

      fileInput("file", label = h3("Histogram Data File input")),
      tableOutput("tabledf"),
      tableOutput("tabledf2"),
      plotOutput("histPlot1")  ))
```

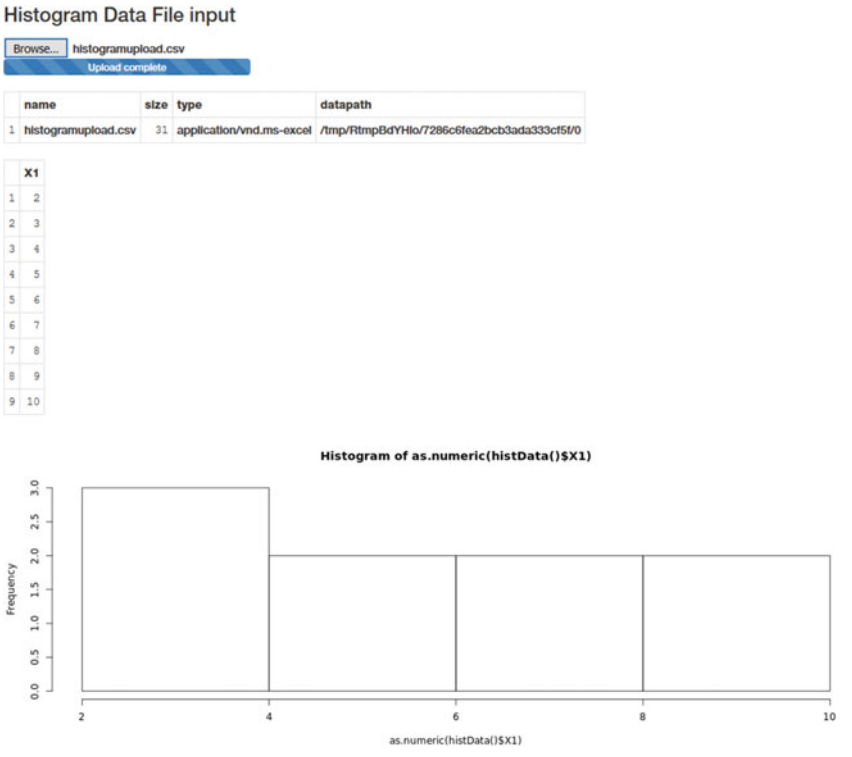


Figure 13-5. *File upload results*

First of all, notice the first table. It contains four variables that are the results of directly accessing the `input$file` variable. This might be counterintuitive based on our other examples, where the user inputs could be directly accessed from their `input$inputId`. However, it does show that our user input is, in fact, a data frame, and the temporary data path is available to access just as any file location might be. Fair warning: If a user uploads another file, since we are in a temp directory, our old file might well be overwritten. Next, notice that our file's actual data is in the second table, and as shown in the histogram, that information is available for any R computations we might choose to code.

Let's next focus on the server side, because this is not as simple as our prior examples. Sometimes, you have input that takes a little more managing than a single numeric or text value. Because that data is input data, it needs a reactive function to be alerted if it has changed. However, you might prefer to use fairly clean code. Fortunately, Shiny offers a basic reactive function named, well, `reactive()`. In the following code, we create our own reactive function to cope with a new scenario, namely, the one we wanted. When a file is uploaded, Shiny alerts our reactive function `reactive()` that `input$file` has changed. The function naturally queries the input to see what it is now, and stores that data frame into the variable `file1`. We would like access to the data inside our CSV file, so we need to call `read.csv()` on that file, which naturally needs to know the file's location. That is stored in the last column, named `datapath`. What we get is an object that we can treat like a data frame later in our code.

```
histData<-reactive({
  file1<-input$file
  read.csv(file1$datapath,header=TRUE ,sep = ",")      })
```

Other than this, we can use our data now inside other reactive functions, familiar ones even. We show one more new reactive function, `renderTable()`, as well as a familiar one, `renderPlot()`. You should also draw your attention to `histData()$X1`. Notice that this has function parentheses, yet elements are still accessed via the familiar `$`. This is built with a reactive function, and as such, we must acknowledge that it is both a function that takes input from the user as well as a data storage object with data to access.

```
output$tabledf2<-renderTable({
  histData()            })

output$histPlot1<-renderPlot({
  hist(as.numeric(histData()$X1))              })
```

That wraps up our analysis of the code. For completeness, we end this section with the entire code, but now you have the techniques to be up and running in Shiny. Before this section ends, we point out that the references section has as its first link a reference site for most or many Shiny functions. If there is something you want to do, check there first. It is quite intuitive. Also of note is that the next section of this chapter shows how to upload an application file to the cloud and get it working live. Finally, notice the `h3()` function inside the `fileInput()` function in the user interface. Shiny has a whole collection of functions that duplicate various common HTML features such as headers. For this chapter, we consider those adiaphora. In Chapter 14, we focus more on making our user interface, for lack of a better word, pretty.

```
library(shiny)

ui <- shinyUI(fluidPage(

      # Copy the line below to make a file upload manager
      fileInput("file", label = h3("Histogram Data File input"), multiple = FALSE),
      tableOutput("tabledf"),
```

```
      tableOutput("tabledf2"),
      plotOutput("histPlot1")

))

# Define server logic required to draw a
server <- shinyServer(function(input, output) {

  output$tabledf<-renderTable({
    input$file
  })

histData<-reactive({
  file1<-input$file
  read.csv(file1$datapath, header=TRUE ,sep = ",")     })

output$tabledf2<-renderTable({
  histData()           })

output$histPlot1<-renderPlot({
  hist(as.numeric(histData()$X1))               })

})

# Run the application
shinyApp(ui = ui, server = server)
```

Notice that in this code, we did not focus on distinctions such as side panels or main panels; instead, we had our user-interface side (both the requested input and the generated output) all in one hodgepodge. It is worth noting that if you have a side panel, a main panel is needed to hold any output that is not hosted by the side panel.

Hosting Shiny in the Cloud

Our last task is to get our files from a local machine to the cloud instance. Remember, in previous chapters, we have used not only `PuTTY` (Tatham, 2016) to install Shiny Server on our cloud, but also `WinSCP` (Prikryl, 2007) to move files up to our cloud. We start `WinSCP`, reconnect, and ensure that on the left, local side we are in our Shiny application folder, while on our right, cloud server side, we are in `/home/ubuntu`. We show the result of that process in Figure 13-6.

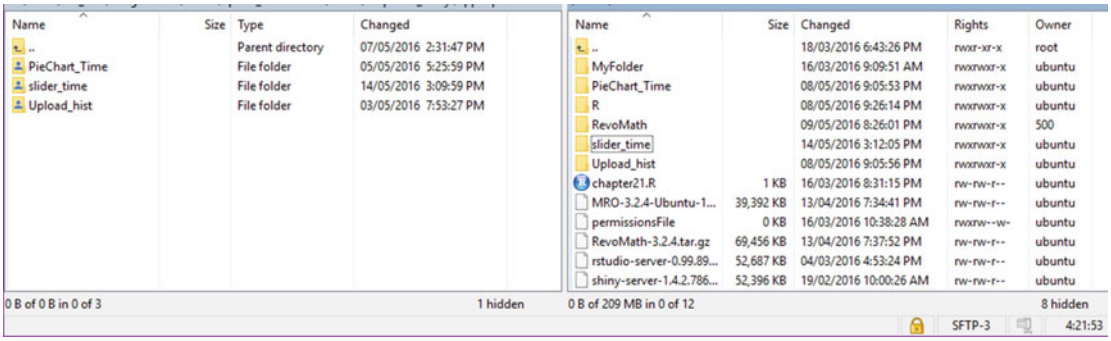


Figure 13-6. *WinSCP with PieChart_Time, slider_time, and Upload_hist Shiny applications all uploaded*

From here, we access PuTTY and run the following code from the command line, one at a time:

```
sudo cp -R ~/PieChart_Time /srv/shiny-server/
sudo cp -R ~/slider_time /srv/shiny-server/
sudo cp -R ~/Upload_hist /srv/shiny-server/
```

Which leaves us with one final step. Recall, back when we first set up our cloud instance, we carefully allowed just a few ports to work for just a few IP addresses. You need to log back into AWS and get into the EC2 Dashboard again. From there, on the left navigation menu, go back to Security Groups and select your AdvancedR group. Once that group is selected, click the Inbound tab and then click Edit ä Add Rule. You want to use a Custom TCP Rule, and add port 3838, and finally lock it to your IP address. Then click Save. The result looks like Figure 13-7.

Type ⓘ	Protocol ⓘ	Port Range ⓘ	Source ⓘ
HTTP	TCP	80	72.18.154.27/32
SSH	TCP	22	72.18.154.27/32
Custom TCP Rule	TCP	8787	72.18.154.27/32
Custom TCP Rule	TCP	3838	72.18.154.27/32
HTTPS	TCP	443	72.18.154.27/32

Figure 13-7. *The AWS Inbound table with Shiny Server port 3838 open to one single local IP address*

You may now verify that your apps work by typing the following URLs into any browser:

```
http://Your.Instance.IP.Address:3838/PieChart_Time/
http://Your.Instance.IP.Address:3838/slider_time/
http://Your.Instance.IP.Address:3838/Upload_hist/
```

Should you wish to allow others or the world to access your applications, change the Inbound source IP address to `0.0.0.0/0` only for port `3838`. It is a security measure not to allow the world access to your SSH port. Of course, if you are now hosting an open-to-the-world port 3838, your server is visible and known. It can be beneficial to have different cloud instances for individual activities. We recommend against having confidential data on public-accessible servers. Again, cloud security is beyond the scope of this book.

Final Thoughts

We seem to have spent some time discussing cloud security. There are two reasons behind this. The first is simply our paranoid natures. However, a fair bit of data can contain confidential information. The ethical use of data, including reasonable safeguards, is important. In recent months, AWS has taken great strides to create safer environments for EC2. As always, security is a matter of reasonableness. The more vital your data, the better steps may be warranted to protect that information.

Those concerns aside, this is truly an unprecedented capability, to allow users to interact with data analytics in such an open fashion. Yes, in our experience, this can lead some data consumers to make uneducated decisions. Still, the benefit of the transparency possible through techniques such as Shiny far outweigh the disadvantages. It is an exciting world, one we are eager to explore.

CHAPTER 14

Shiny Dashboard Sampler

This chapter is not required in order to understand other chapters in this book. While we introduce some new techniques, what you primarily find here is one entire dashboard sample ready to be modified to suit your needs.

Just what does this sampler do? It takes the applications we used in the preceding chapter and presents them in an engaging, interactive format. It is important to keep in mind that while we often present these applications with a graphical output, any R process can be done to these data. The goal is to allow information consumers the capability to naturally interact with live data, whether those consumers are research reviewers, next-level directors, or board members.

Similarly to the last chapter, we introduce this on Windows inside `RStudio` (RStudio Team, 2015). We provide both a framework to understand this dashboard, as well as some pro tips. It is important to see this sampler as a creativity incubator of sorts. It is our hope that you see something you like, and more vitally, realize just how you can best showcase your data.

We will start with the underlying structure of a dashboard.

A Dashboard's Bones

Recall our littlest `shiny` (Chang, Cheng, Allaire, Xie, and McPherson, 2016) application from the preceding chapter:

```
library(shiny)

# Define user interface
ui <- shinyUI(fluidPage())

# Define server
server <- shinyServer(function(input, output) {})

# Run the application
shinyApp(ui = ui, server = server)
```

This code was good to look at from the beginning, because it highlights how all Shiny applications comprise a user interface, and a server side where the R code logic lives—and these are both run as an R process themselves (possibly on a cloud server). All we intend to add to this structure is some code on the user-interface side. This should make sense, because a dashboard is more about user interface than any

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_14

new code logic. The goal is to make analytics and statistical inferences readily available to users. There is a new library to install for this, but it is just one, `shinydashboard` (Chang, 2015). From the command line of RStudio, go ahead and run the following code:

```
install.packages("shinydashboard")
```

With this new package installed, we may now take a look at the littlest Shiny *dashboard* application. Dashboards include a header, a sidebar with various menu items, and the main body in which the different objects of the applications exist. Compare and contrast this dashboard code layout to the single application code. The two versions are almost the same. After the code, see what they generate in Figure 14-1.

```
library(shiny)
library(shinydashboard)

# Define user interface
ui <- dashboardPage(
                dashboardHeader(),
                dashboardSidebar(),
                dashboardBody()
               )

# Define server
server <- shinyServer(function(input, output) {})

# Run the application
shinyApp(ui = ui, server = server)
```

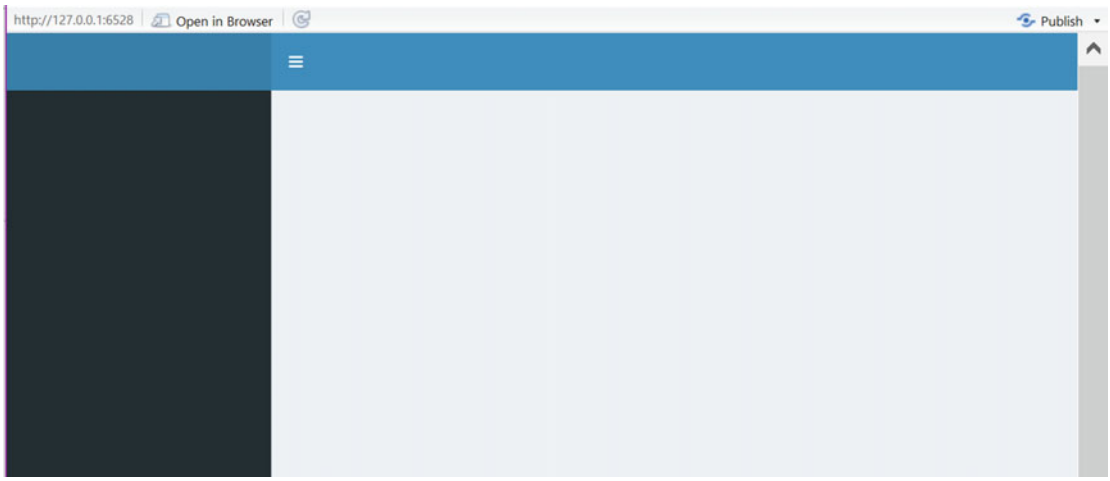


Figure 14-1. *The littlest Shiny dashboard is quite empty*

As you can see in the preceding code, we have three functions: `dashboardHeader()`, `dashboardSidebar()`, and `dashboardBody()`. These provide a nice structure for our dashboard. These all reside inside `dashboardPage()` as the first three of five formal arguments for the page function. The final two formals include `title` (which defaults to `NULL`) as well as `skin` (which takes on one of `blue`, `black`, `purple`,

green, red, or yellow color). As you can see for yourself with the formals(dashboardPage) command, the page function is indeed simple enough. The title argument controls the heading in the browser itself (or on the browser tab, if multiple tabs are open in most popular browsers). The skin color options control the overall color palette of the dashboard. Now, the first three formals are functions in their own right and become quite complex. Nevertheless, we will get to all three in turn.

Dashboard Header

The dashboardHeader() function also takes a title formal that adds user-defined text into the header area as well as a titleWidth argument controlling the width of that text. The header also accepts several other Shiny user-interface-style functions to control page navigation along with a handful of more specialized functions that allow for more interactive navigation. In this chapter, and indeed this book, we essentially ignore these functions to promote shiny and shinydashboard as ways of serving already familiar R code to data consumers rather than learning too many new features all at once. In fact, the last formal we mention for this function is disable, which defaults to FALSE and on TRUE hides the header entirely. In our sampler, we will preserve the header in a mostly tabula rasa state.

Dashboard Sidebar

Moving on to the dashboardSidebar() function, the only explicitly called-out formals are again a disable option along with a width argument. However, there are more functions possible here beyond the called-out ones. The first thing to know is that any of the user-interface functions from shiny works in the sidebar. Thus, if it makes sense to place a slider bar or drop-down input functions in your menu, that is possible. There are, however, some functions unique to the dashboard library that make sense to place inside the sidebar area.

The first of these is the sidebarMenu() function. Much like dashboardSidebar() itself, this function primarily takes other inputs and handles the organization of them in an efficient fashion. Of particular note is the menuItem() function, which is called from inside sidebarMenu(). In this sampler, we house each of the applications created in the previous chapter in their own call to menuItem(). As you can see in the following code, this function has several formals:

```
formals(menuItem)
$text

$...

$icon
NULL

$badgeLabel
NULL

$badgeColor
[1] "green"

$tabName
NULL

$href
NULL
```

```
$newtab
[1] TRUE

$selected
NULL
```

In the preceding function and formal arguments, the `text` takes a text string in quotes that provides the text information or link name for that particular item on the menu. The `icon` formal is given a name for a `Font-Awesome` icon. We provide a link to the online library, showcasing all possible icon choices in the references section of this chapter. We may also switch to a different icon library; we save such explorations for you. The `badgelabel` call puts an inline badge on the far right of the sidebar, which fits a short bit of text. Badges do a good job of calling viewers' attention to a particular menu item and, consequently, to the underlying application. Many of these formals are somewhat optional; however, there must be either a `tabName` linked to a related `tabItem()` in the dashboard's body or there must be a `URL` given to the `href` command. If a `URL` is given rather than a `tabName`, then `newtab` can be changed from a default of `TRUE` to `FALSE` should you wish your viewer to leave your site when visiting that page. Finally, it is possible to force the default to a particular tab, away from the first `menuItem()`, by setting selected to `TRUE`.

As we turn our attention to the main body of our dashboard, we remind you that the sidebar can also receive any Shiny function—such as the `img()` function which points to an image that must be contained in a folder named `www` inside your dashboard application's folder. We show a code snippet (which should not be run yet) of our sampler's menu along with the result in Figure 14-2. The names of the first three menu items should be familiar from the prior chapter.

```
sidebarMenu(
      menuItem("Pie Charts!", tabName = "PieChart_Time", icon = icon("pie-chart")),
      menuItem("Use a Slider", tabName = "slider_time", icon = icon("sliders"), badgeLabel =
      "New"),
      menuItem("Upload a Histogram File", tabName = "Upload_hist", icon = icon("bar-
      chart")),
      menuItem("Labour Data Dates", tabName = "labour", icon = icon("cloud-download"),
      badgeLabel = "Live")
    )
```

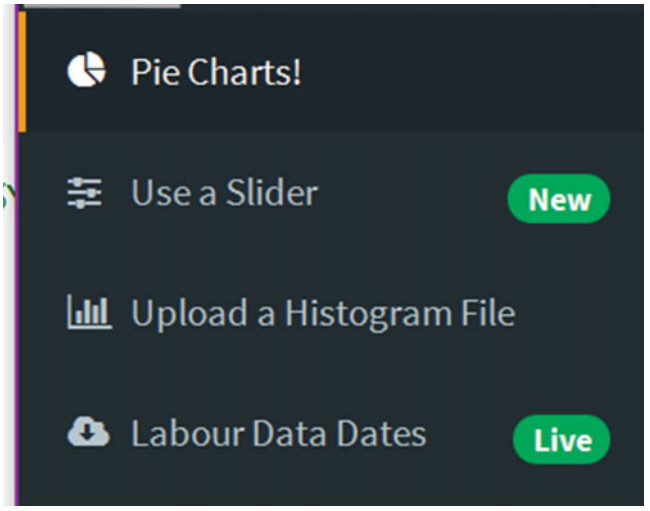


Figure 14-2. *The result of some sidebarMenu() and menuItem() functions*

Dashboard Body

As you can see in the preceding code, each menu item has `tabName`. We will now discuss the code inside the `dashboardBody()` function that will catch that `tabName`. Take a look at the little, although not the littlest, Shiny dashboard in the following code and Figure 14-3, and then we will walk through the new lines.

```
library(shiny)
library(shinydashboard)

# Define user interface
ui <- dashboardPage(
    dashboardHeader(),
    dashboardSidebar(sidebarMenu(menuItem("Demo", tabName = "First"))),
    dashboardBody(tabItems(tabItem(tabName = "First", "test"))),
    skin = "yellow")

# Define server
server <- shinyServer(function(input, output) {})

# Run the application
shinyApp(ui = ui, server = server)
```

Figure 14-3. *A dashboard with a menuItem() in the sidebar and a tabItem() in the body. Notice the matching First.*

Many items might be placed inside the body of our dashboard. However, if we want to use the tabs that we coded back in the sidebar section, we do a general call to `tabItems()`. This function takes no arguments except individual `tabItem()` function calls; each of those `tabItems()` needs to have a unique `tabName` matching a unique `tabName` from the `sidebarMenu()`.

What goes inside each `tabItem()`? Well, just as in Shiny, there is a benefit to making the layout fluid enough that your dashboard can cope with various-sized browser windows. Thus, the common function `fluidRow()` makes a reappearance. Recall that `fluidRow()` is set up to have a row width of at most 12. Remembering that fact, keep a sharp lookout at the default widths for each of the three box objects we discuss to fill those rows. Now, there are more than these three objects; again, our goal is to get you quickly to the point where you may serve your data via the cloud to your users or stakeholders.

The most basic input element to go in a row is `box()`. Inside this function, we place any of the application code we used in the preceding chapter to either solicit input from users or to output reactive content. However, `box()` also takes nine specific arguments. Naturally, each `box()` may take a `title` value, a user-defined text string that should succinctly describe the contents of the box or the action required. At the bottom of the box, there is a `footer`, which may also take a text string. Boxes also have a `status` that may

be set to take on values of `primary`, `success`, `info`, `warning`, or `danger`. Each of those status values has an associated color that is part of the default Cascading Style Sheets structure that makes `shinydashboard` look rather pretty. The colors are fairly consistent, although they may have some variance depending on which color you selected for the whole dashboard. The default value for `solidHeader` is `FALSE`, although this, of course, may be changed to `TRUE`. The positive choice simply enhances the color of the status to become a narrow banner background to the `title` text. Should you wish the main area of your box to have a specific color, `background` can be set from approximately 15 valid colo choices. Any input code and any output code lives on top of that `background`, as the name suggests. The default `width` for a box is `6`, so two boxes fill up a row unless you take steps to narrow them. Failure to keep the total `fluidRow()` width of `12` in mind can lead to some odd layouts. Box `height` can also be set to a fixed height. This can be helpful if you require very even rows. Finally, boxes are `collapsible` and may be already `collapsed`, although both options default to `FALSE`. Figure 14-4 shows some results of these various settings.

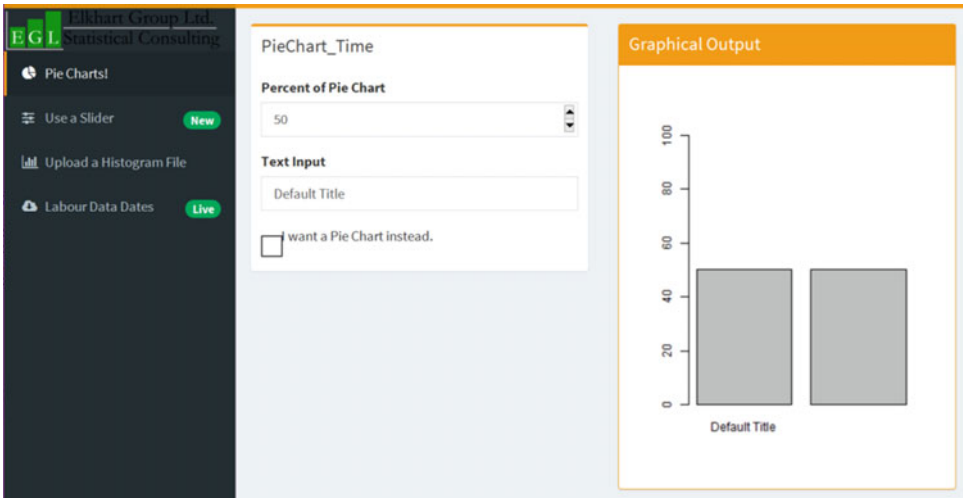


Figure 14-4. *A familiar application in two boxes with various options—see Complete Sampler Code for details*

Another type of box is `valueBox()`, which takes six formal arguments. The first is `value`, which is usually a short text string or value, as shown in Figure 14-5. The `valueBox()` also takes a `subtitle` argument, which is again a text string. You can choose an `icon`, although the default value is `NULL`. These boxes come with a default `color` of `aqua` and are rather short with a default `width` of only `4`. Finally, `valueBox()` also takes a value for `href`, which defaults to `NULL`. We show the code we used in our sampler's value box here:

```
valueBox(endYear-startYear, "Days of Data", icon = icon("calendar"))
```

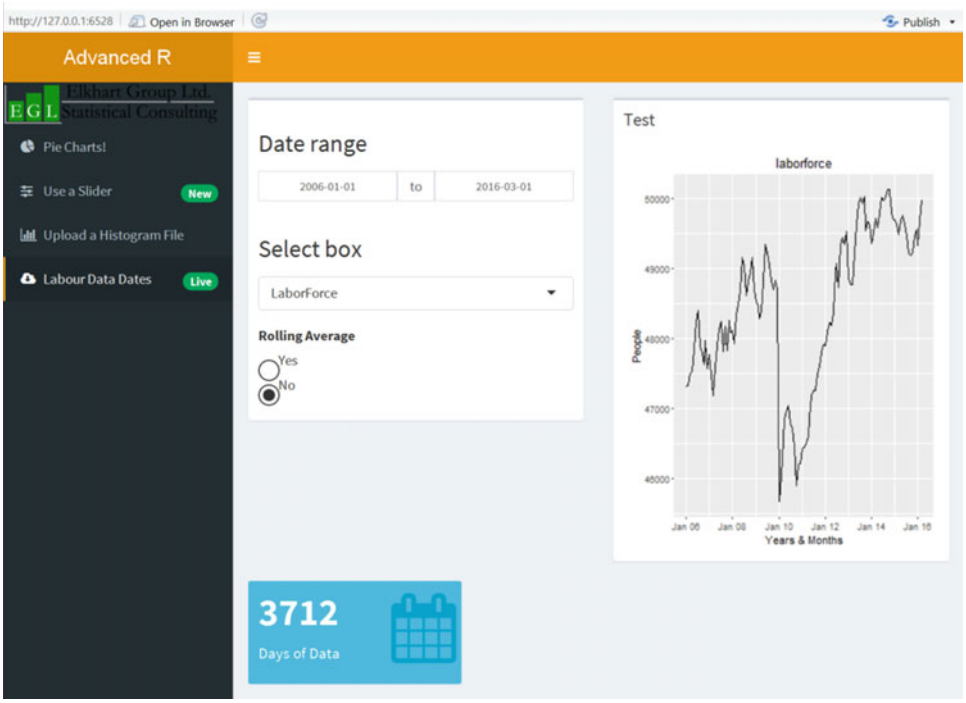


Figure 14-5. *Sample valueBox() showing the number of days of data that were pulled live from a website*

We finish out our discussion of the `dashboardBody()` boxes with a box similar to the `valueBox()`: the `infoBox()`. This box comes with a `title` that takes a text string, a `value` to highlight and display in the box, optional `subtitle` text, and a default `icon` of a bar chart. It also features defaults for `color` of `aqua`, `width 4`, and a `NULL href`. What is different is that the `icon` is smaller, and there is a `fill` option, which is usually `FALSE`, that can change the way the background color displays. We do not show this box, but it does look quite similar to the small box in Figure 14-5. In our experience, these boxes are not significantly different other than in their appearance.

Dashboard in the Cloud

We do want to serve this dashboard into our cloud instance, and this is a *slightly* more complex set of code. Again, we created this in a standard Shiny folder by using RStudio, so it comes in its own file. The file structure on Windows looks like Figure 14-6. There are three files in our `shinydashboard_cloud` folder. Note the `www` folder, which hosts our sampler's image, followed by the `app.R` file, which holds our entire Shiny application as well as our CSV file.

Figure 14-6. *The file structure for shinydashboard_cloud*

We use WinSCP to upload our files and folders to our Ubuntu cloud instance, as shown in Figure 14-7.

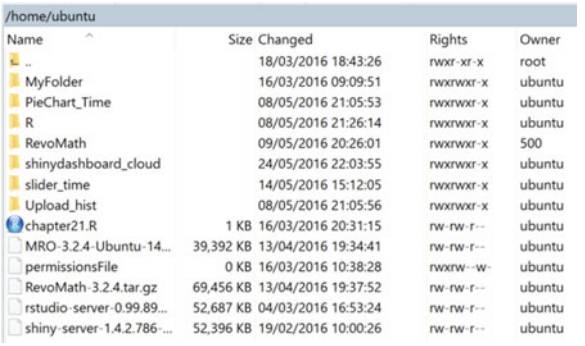


Figure 14-7. *WinSCP upload folder view*

Logging into PuTTY, we do have some work to do. Our sampler uses some new libraries, and, of course, the file needs to be moved from our user's folder to where the Shiny server expects it to live. For brevity, we install only the packages we have not yet installed on our cloud instance.

The first of the new packages is XML (Lang, D. et al., 2016), and this package provides the ability to read in an HTML table live from the Web. Various government and nonprofit agencies store data in such tables for easy access by the public. We also install the zoo package (Zeileis and Grothendieck, 2005), which provides convenient functions for time series. Already familiar packages are ggplot2 (Wickham, 2009) and data.table (Dowle, Srinivasan, Short, and Lianoglou, 2015), which handle graphical plotting and provide a more advanced data frame structure.

The following commands run one at a time on our instance installation for all users' five new libraries and then copy the files so they are accessible via a browser. We also install some software onto Ubuntu that is required to allow the XML package to run. For readability, we group some of the following commands together. They should be run one line at a time in PuTTY, and some text prints to the screen after each command.

```
sudo su - -c "R -e \"install.packages('shinydashboard', repos='https://cran.rstudio.
com/')\""

sudo apt-get install libcurl4-openssl-dev libxml2-dev
sudo su - -c "R -e \"install.packages('XML', repos='https://cran.rstudio.com/')\""

sudo su - -c "R -e \"install.packages('zoo', repos='https://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('data.table', repos='https://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('ggplot2', repos='https://cran.rstudio.com/')\""

sudo cp -R ~/shinydashboard_cloud /srv/shiny-server/
```

This completes our work with Shiny and the Shiny dashboard. Your dashboard should be live at http://Your.Instance.IP.Address:3838/shinydashboard_cloud/ and viewable from your web browser. We have a few more comments to make before we close out the chapter.

First, we highly suggest that you visit the Apress site for this textbook and download the code bundle for this chapter. It contains the complete, working files and folders shown in Figure 14-6 and uploaded in Figure 14-7. There is no need to manually type all this code to see whether a dashboard might be something you would like to have.

Second, we have in no way throughout this chapter given our full sample code. We saved that for the end to keep it all in a single, cohesive whole and to focus your attention on the specific structure you want to be looking for in that sampler code.

Third, we remind you that if you have followed our security-minded advice about inbound policy settings on the AWS servers, you may need to change those settings to allow more people than just yourself to view your final dashboard project. Remember, the setting you need to change on the inbound side is to allow port `3838` to be accessed from `0.0.0.0/0`, which is the entire Internet. Keep in mind that the entire Internet is then able to access your dashboard. If you have sensitive data, or even information that just does not need to be out and about, please consider partnering with an IT professional or at least an IT power user.

Finally, we hope you enjoyed this chapter and the results as much as we have. Be sure to explore the links in the "References" section. You will find valuable information ranging from additional functions, a whole list of icons, a whole second icon library, and the very latest in `shiny` and `shinydashboard` package information. These packages are updating fairly regularly, and new features can often provide awesome results.

Complete Sampler Code

```
library(shiny)
library(shinydashboard)
library(XML)
library(zoo)
library(data.table)
library(ggplot2)

#this code will run just once each time a session is instantiated - there may be multiple
viewers in the same session.

slider_data<-read.csv("slider_data.csv", header = TRUE, sep = ",")
Phase1    <- slider_data[,2]
Phase2    <- slider_data[,3]
Phase3    <- slider_data[,4]
Phase4    <- slider_data[,5]

bls <-readHTMLTable("http://data.bls.gov/timeseries/LAUMT484702000000006?data_tool=XGtable",
stringsAsFactors = FALSE)

bls <- as.data.table(bls[[2]])
bls[, names(bls):= lapply(.SD, function(x) {
  x <- gsub("\\(.*\\)", "", x)
  type.convert(x)
  }
  )]

setnames(bls,names(bls), gsub("\\s","",names(bls)))
bls[, Date := as.Date(paste0(Year, "-", Period, "-01"), format="%Y-%b-%d")]
startYear<-min(bls$Date)
endYear<-max(bls$Date)
```

```
### We now define the user interface to be a dashboard page
ui <- dashboardPage(
  dashboardHeader(title = "Advanced R"),

  dashboardSidebar(
    img(src = "egl_logo_full_3448x678.png", height = 43, width = 216),
    sidebarMenu(
      menuItem("Pie Charts!", tabName = "PieChart_Time", icon = icon("pie-chart")),
      menuItem("Use a Slider", tabName = "slider_time", icon = icon("sliders"),
                                                        badgeLabel = "New"),
      menuItem("Upload a Histogram File", tabName = "Upload_hist", icon = icon("bar-
      chart")),
      menuItem("Labour Data Dates", tabName = "labour", icon = icon("cloud-download"),
                                                        badgeLabel = "Live")
    ) ),

dashboardBody(
    tabItems(
      # PieChart_Time Content which should be familiar
      tabItem(tabName = "PieChart_Time",
              fluidRow(
                box( title = "PieChart_Time", status = "warning",
                 numericInput("pie", "Percent of Pie Chart", min = 0, max = 100,  value = 50),

                            textInput("pietext", "Text Input", value = "Default Title",
                            placeholder = "Enter Your Title Here"),

                            checkboxInput("pieChoice",
                                          "  I want a Pie Chart instead.", value = FALSE)
                ),

                box( title = "Graphical Output", solidHeader = TRUE, status = "warning",
                  plotOutput("piePlot")))),

      # Slider Tab Content which should be familiar
      tabItem(tabName = "slider_time",
              h2("Training Programme Results"),
              box( title = "Control the Academic Year", status = "primary", solidHeader =
              TRUE,
                        sliderInput("ayear", "Academic Year:", min = 2014, max = 2017, value
                        = 2014, step = 1,
                                    animate = animationOptions(interval=600, loop=T))),

              box(plotOutput("barPlot"))),

##Histogram from an Uploaded CSV which should be familiar
      tabItem( tabName = "Upload_hist",

               fluidRow(
                        box(   title = "File Input",
```

```
                    # Copy the line below to make a file upload manager
                    fileInput("file", label = h3("Histogram Data File input"),
                              multiple = FALSE)
                  ),
                  box( title = "Data from file input", collapsible = TRUE,
                    tableOutput("tabledf"))),

              fluidRow(
                box(tableOutput("tabledf2")),
                box( background = "blue",
                  plotOutput("histPlot1"))
              )

            ),

###This Labour plot Tab is new and uses data pulled fresh from the web each time an R
instance ##is created.
      tabItem(tabName = "labour",

              fluidRow(
                box( dateRangeInput("labourDates", min = startYear,max = endYear,
                                    start = startYear, end = endYear,
                                    label = h3("Date range")),

                     selectInput("labourYvalue", label = h3("Select box"),
                                 choices = list("LaborForce" = 3,
                                                "Employement" = 4,
                                                "Unemployement" = 5),
                                 selected = 3),

                     radioButtons("labourRadio", "Rolling Average", selected = FALSE,
                                  choices = list("Yes"=TRUE, "No"=FALSE))),
                box(title = "Test",  plotOutput("labourPlot"))),

              fluidRow( valueBox(endYear-startYear, "Days of Data", icon =
              icon("calendar"))))))),
  title = "Dashboard Sampler",  skin = "yellow")

#### Define server logic required to draw a histogram#####
server <- shinyServer(function(input, output) {

  output$piePlot <- renderPlot({
# generate Pie chart ratios based on input$pie from user
    y <- c(input$pie, 100-input$pie)

# draw the pie chart or bar plot with the specified ratio and label
    if(input$pieChoice == FALSE){
      barplot(y, ylim = c(0,100), names.arg = c(input$pietext, paste0("Complement of ",
      input$pietext)))
```

```
    }else{
      pie(y, labels = c(input$pietext, paste0("Complement of ", input$pietext)))}
  })

  output$barPlot <- renderPlot({
    # Count values in each phase that match the correct date.
    cap<-input$ayear*100
    x <- c(sum(Phase1<cap), sum(Phase2<cap),  sum(Phase3<cap),  sum(Phase4<cap))

    # draw the bar plot for the correct year.
    barplot(x,
            names.arg = c("Phase I", "Phase II", "Phase III", "Fellows"),
            col = c("deeppink1","deeppink2","deeppink3","deeppink4"), ylim=c(0,50))
  })

  ####Here is where the input of a file happens
  output$tabledf<-renderTable({  input$file  })

  histData<-reactive({
    file1<-input$file
    read.csv(file1$datapath,header=TRUE ,sep = ",")
  })

  output$tabledf2<-renderTable({  histData()  })

  output$histPlot1<-renderPlot({    hist(as.numeric(histData()$X1))  })

  ##############labour plot####################
  output$labourPlot<-renderPlot({
    xlow<-input$labourDates[1]
    xhigh<-input$labourDates[2]
    blsSub <- bls[Date<=xhigh&Date>=xlow]
    yValue <- names(blsSub)[as.numeric(input$labourYvalue)]

    if(input$labourRadio == TRUE && nrow(blsSub)>=4){  blsSub[,(yValue):=rollmean(get(yValue),
    k=3, na.pad = TRUE)] }

    ggplot(data = blsSub, mapping = aes_string("Date", yValue)) +
      geom_line()+
     xlab("Years & Months")+
     ylab("People")+
     xlim(xlow, xhigh)+
      ggtitle(yValue)+
      scale_x_date(date_labels = "%b %y")
  }) })

# Run the application
shinyApp(ui = ui, server = server)
```

References

- https://rstudio.github.io/shinydashboard/
- http://fontawesome.io/icons/
- http://shiny.rstudio.com/reference/shiny/latest/
- http://docs.rstudio.com/shiny-server/
- http://shiny.rstudio.com/gallery/
- https://cran.r-project.org/web/packages/shiny/shiny.pdf

CHAPTER 15

Dynamic Reports and the Cloud

In the preceding chapters, you saw how Shiny delivers interactive environments based on dynamic data. Dynamic reports and the live dashboards have many similarities. On the other hand, this chapter, because the reports end up as PDFs, is less interactive. Reports are a fact of life in many fields, and stakeholders tend to require snapshots in time rather than fully interactive environments. Through the `knitr` and `rmarkdown` packages, we create documents (for example, PDF, HTML, or Microsoft Word) based on data input. For regular reports that build on continuously changing data, yet that have the same structure overall, this is a great time-saver.

The `rmarkdown` package (Xie, 2015) adds the capability to embed R processes into such documents, allowing one-click analytics that are beautifully formatted using a variety of styles. The LaTeX interface achieves almost any formatting required, although there are options besides PDF. From boardroom reports on enrollment based on live pulls from a student information system database, to strategic plan action-sheets drawn from key performance indicators, dynamic reports work best when variable input needs shaping into standardized output. This is also facilitated via `knitr` and `formatR` packages (Xie, 2015).

Additionally, in this chapter, we use the `evaluate` package (Wickham, 2016) and `tufte` package (Xie, 2016). These packages allow us to organize the layout and format of our date more elegantly. We also use the `ltm` package (Rizopoulos, 2006) for the psychometric tests.

Our goals for this chapter are to enhance our software environment both on our local and cloud machines to allow the dynamic documents to compile, explore the structure of standalone `rmarkdown`, develop a Shiny page that allows a user to upload data and then download a PDF report, and finally to push this all to our cloud instance. We start with needed software.

Needed Software

We already have a great deal of the software needed, namely R, the Shiny packages (Chang, Cheng, Allaire, Xie, and McPherson, 2016), PuTTY, and our cloud instance on Ubuntu. We need a handful of new packages installed—both on the local machine as well as the cloud server.

Local Machine

First, in your R console, it is important to run the following code independent of any R file:

```
install.packages("rmarkdown")
install.packages("tufte")
install.packages("knitr")
```


M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1_15

```
install.packages("evaluate")
install.packages("ltm")
install.packages("formatR")Dynamic documentlocal machine
```

After those packages install, we turn our attention to installing MiKTeX (Schenk, 2009), which allows the creation of PDF files. A brief visit to the MiKTeX site (http://miktex.org/download) allows you to obtain either the recommended basic installer (which is 32-bit) or the other basic installer (which is 64-bit). We use the 64-bit version for our systems, although in this chapter there is not likely to be a major difference between the versions. After download, accept all the defaults during installation. After installation, there may be updates to the MiKTeX package. To perform these updates, go to your start files. In a folder titled MiKTeX, you'll find an option for Update near the end. Follow the default options and then select all packages to update to the latest. While not required, we do recommend a restart after this installation.

Cloud Instance

Using PuTTY, access your cloud instance console. We need to run each line of code that follows individually at the console. Recall from an earlier chapter that these install the packages for all users of that server, not just your user. This is key to allow the Shiny server access to the correct packages.

```
sudo su - -c "R -e \"install.packages('rmarkdown', repos = 'http://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('tufte', repos = 'http://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('knitr', repos = 'http://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('ltm', repos = 'http://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('formatR', repos = 'http://cran.rstudio.com/')\""
sudo su - -c "R -e \"install.packages('data.table', repos = 'http://cran.rstudio.com/')\""
```

The preceding code does not take very long. However, installing texlive (the Ubuntu solution for LaTeX used instead of MiKTeX), may take some time. It also demands a fair bit of disk space, so be sure you have room. In our trials, the full version is required.

```
sudo apt-get install texlive-full
```

After this, we recommend a sudo reboot to restart your cloud instance. This is a good place to do that and then close PuTTY for a bit. While your server restarts, you may use your local machine to explore dynamic documents.

Dynamic Documents

It is not required to link dynamic documents to the Internet in any way. Indeed, most reports may be internal. Inside RStudio, selecting File ➤ New File ➤ R Markdown opens a wizard. Notice that there are options for documents (for example, HTML, PDF, or Word) as well as presentations (for example, HTML or PDF). A title is customary, as is listing the author(s). Of course, RStudio is not required; creating a new file in R with the .Rmd suffix suffices.

We intend to create a hybrid approach to this introduction to dynamic documents. That is, we first give a straightforward and informative example of what such code looks like, and then we follow up with a more in-depth analysis of each piece of that code. We start with a new file titled `ch15_html.Rmd` and fill it with the following code:

````
---
title: "ch15_html"
author: "Matt & Joshua Wiley"
date: "18 September 2016"
output: html_document
---

```{r echo=FALSE}
##notice the 'echo=FALSE'
##it means the following code will never make it to our knited html
library(data.table)
diris <- as.data.table(iris)
##it is still run, however, so we do have access to it.
```

#This is the top-level header; it is not a comment.

This is plain text that is not code.

If we wish *italics*  or **bold**, we can easily add those to these documents. Of course, we
may need to mention `rmarkdown` is the package used, and inline code is nice for that. If
mathematics are required, then perhaps x~1~^2^ + x~2~^2^ = 1 is wanted.

##calling out mathematics with a header 2
On the other hand, we may need the mathematics called out explicitly, in which case  $x^{2}
+ y^{2}= \pi$ is the way to make that happen.

###I like strikeouts; I am less clear about level 3 headers.
I often find when writing reports that I want to say something is ~~absolutely foolish~~
obviously relevant to key stakeholders.

######If you ever write something that needs Header 6
Then I believe, as this unordered list suggests:

* You need to embrace less order starting now

    - Have you considered other careers?

```{r echo=FALSE}
summary(diris)
```

However, notice that one can include both the code and the console output:

```{r}
hist(diris$Sepal.Length)
```
````

The results of knitting the preceding code are shown in Figure 15-1.

ch15_html

Matt & Joshua Wiley

18 September 2016

This is the top level header; it is not a comment.

This is plain text that is not code.

If we wish *italics* or **bold** we can easily add those to these documents. Of course, we may need to mention `rmarkdown` is the package used, and inline code is nice for that. If mathematics are requried, then perhaps $x_1^2 + x_2^2 = 1$ is wanted.

calling out mathematics with a header 2

On the other hand, we may need the mathematics called out explicitly, in which case $x^2 + y^2 = \pi$ is the way to make that happen.

I like strikeouts, I'm less clear about level 3 headers.

I often find when writing reports that I want to say something is ~~absolutely foolish~~ obviously important to key stakeholders.

If you ever write something that needs Header 6

Then I believe, as this unordered list suggests:

- You need to embrace less order starting now
    - Have you considered other careers?

```
##   Sepal.Length    Sepal.Width     Petal.Length    Petal.Width
##  Min.   :4.300   Min.   :2.000   Min.   :1.000   Min.   :0.100
##  1st Qu.:5.100   1st Qu.:2.800   1st Qu.:1.600   1st Qu.:0.300
##  Median :5.800   Median :3.000   Median :4.350   Median :1.300
##  Mean   :5.843   Mean   :3.057   Mean   :3.758   Mean   :1.199
##  3rd Qu.:6.400   3rd Qu.:3.300   3rd Qu.:5.100   3rd Qu.:1.800
##  Max.   :7.900   Max.   :4.400   Max.   :6.900   Max.   :2.500
##        Species
##  setosa     :50
##  versicolor:50
##  virginica :50
##
##
##
```

However, notice that one can include both the code and the console output:

```
hist(diris$Sepal.Length)
```

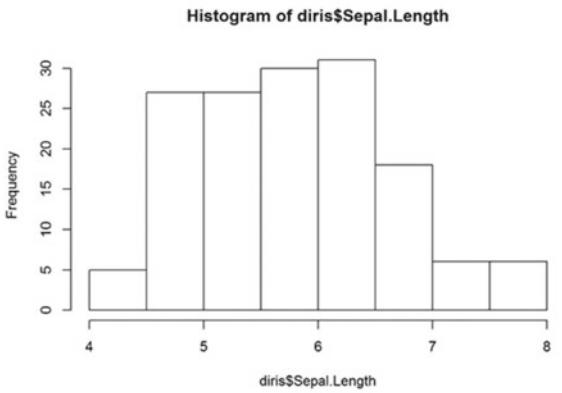


Figure 15-1. *The HTML output of a simple dynamic document*

As you can imagine, it is the summary data or the graph that is easily regenerated based on any input. In fact, one of the authors uses dynamic documents to create case studies (and a case study key) for students in an introductory statistics course. It is easy enough to generate pseudorandom data with different seeds that allow each student to have a unique data set (and several topics). It also allows for solution keys for fast marking.

Now that you have an overview of `Rmarkdown`, let's explore the previous code in sensible blocks. The first block is *YAML*, which stands for *Yet Another Markup Language* or possibly the recursively defined `YAML Ain't Markup Language` (YAML, 2016) header. At its simplest, it may contain only information about the title, author, date, and the output type. Here, for this document, we selected `html_document`. However, as stated previously, there are more options such as `pdf_document`, `word_document`, `odt_document`, `rtf_document`, and `beamer_presentation` (to name a few—there are in fact more). These types create, respectively, HTML pages, PDFs, Microsoft Word files, OpenDocument or rich text, and Beamer PDF slides. Both Beamer and PDFs require the `MiKTeX` installation we did earlier. To change the type of document compiled, simply change the output! It is worth noting that the first time you use `MiKTeX`, it may take longer to compile, as there may need to be a package or two installed behind the scenes. This does not take any user intervention; it simply takes time. It is also worth noting that if you use `beamer_presentation`, then what can and cannot fit on one slide needs to be considered. In that case, a carriage return followed by `----`, followed by another carriage return, signals new slides in your deck.

```
---
title: "ch15_html"
author: "Matt & Joshua Wiley"
date: "18 September 2016"
output: html_document
---
```

Headers can, be more complex than this. R code can be injected inline, using `` `r code goes here` `` formatting. Thus, we could change the date to be more dynamic by swapping out `date: "18 September 2016"` for `date: "` r Sys.Date()`"` in our code. Dozens of render options work with the YAML header, and wiser heads than ours have created templates that use variants of those headers as well as set several nice options. As in Shiny, there is a nearly limitless ability to customize precisely how a document outputs and displays.

The next region of our markdown code is a code chunk. Code chunks start with ```` ```{r } ```` and end with ```` ``` ````. Between those lives your code. Now, inside the `{r }` there are several useful options. If you have part of your code that is intensive to create and mostly static, you may want to set `cache = TRUE` instead of the default `FALSE`. This stores the results. We use `echo=FALSE` often, which prevents the code itself, yet not the results of that code, from displaying. Notice that in the first chunk that follows, there is no sign in our final document that the code ran. However, the second chunk still shows the results of `summary(diris)`.

````
```{r echo=FALSE}
library(data.table)
diris <- as.data.table(iris)
```

```{r echo=FALSE}
summary(diris)
```
````

For code chunks that involve plots or figures, there are many options too. The `fig.align` option may be set to `right`, `centre`, or `left`. There are height and width settings for plots with `fig.height` or `fig.width` that default to inches.

````
```{r fig.align="right"}
hist(diris$Sepal.Length)
```
````

Finally, there is the text itself, which we do not repeat here because that is of less interest. We do note that code can be written inline as well as just as in the header. You'll see that this option becomes very useful later, when we want to provide different narrative text depending on our computation results.

Dynamic Documents and Shiny

We turn our attention now to providing not only a dynamic document but one that others may control via their dynamic input. What we build is a simple Shiny environment that allows for a user to upload a CSV file with quiz score data for students. The data is coded as follows: the columns represent specific questions on the quiz; the rows represent individual students; and the quiz is multiple choice, with `1` coded in for correct responses, and `0` coded in for incorrect options.

As in the preceding section, we look at all our code for each of the three files it takes to make this work. Then, we break that code down line by line. For readers already comfortable with Shiny, please feel free to skip to the last portion. We build this code in a folder named `Chapter15`. Later, when we upload to our Ubuntu server, we upload that entire folder.

server.R

On the Shiny server side, we call our libraries and then build the systems needed to upload our CSV file, provide confirmation for users that their file is uploaded, read that file into R, pass the needed information along to our markup document, and then allow our users to download that document as a PDF. It takes just the following 55 lines of code:

```
##Dynamic Reports and the Cloud

library(shiny)
library(ltm)
library(data.table)
library(rmarkdown)
library(tufte)
library(formatR)
library(knitr)
library(evaluate)

  function(input, output) {

    output$contents <- renderTable({

      inFile <- input$file1

      if (is.null(inFile))
        return(NULL)

      read.csv(inFile$datapath, header = input$header,
               sep = input$sep, quote = input$quote)
    })

    scores <- reactive({
      inFile <- input$file1
```

```
    if (is.null(inFile))
      return(NULL)

    read.csv(inFile$datapath, header = input$header,
             sep = input$sep, quote = input$quote)
  })

  output$downloadReport <- downloadHandler(
    filename = function() {
      paste('quiz-report', sep = '.', 'pdf')
    },

    content = function(file) {
      src <- normalizePath('report.Rmd')

      owd <- setwd(tempdir())
      on.exit(setwd(owd))
      file.copy(src, 'report.Rmd', overwrite = TRUE)

      knitr::opts_chunk$set(tidy = FALSE, cache.extra = packageVersion('tufte'))
      options(htmltools.dir.version = FALSE)

      out <- render('report.Rmd')
      file.rename(out, file)
    }    )  }
```

The preceding code started with a call to all our libraries, and then quickly got into Shiny's `function(input, output) {}` server-side wrapper. After that, there are just three main blocks of code. In turn, we look at each of these, starting with the `renderTable({})` call. This generates output used in our user-interface side. In particular, it is creating a table. It is an interactive function, and when a user uploads a file on the user side, Shiny alerts this function that one of its inputs has changed. This triggers a run of the code. From the side of entry, `input$datapath` is the location on the server where Shiny stores an uploaded file. Provided there is, in fact, a file, `read.csv` can access the file's data path, and also gets information about whether the data has a header, what type of separator was used, and some other information set by the user. In fact, much more could be asked of the user that may allow for more customized analytics. Here, we keep the example simple yet efficacious. The result of this read creates a table and stores that table inside the output variable named `contents`. This has a net effect of Shiny now alerting the user-interface side that there is possible output. We see where that output goes later in this chapter.

```
output$contents <- renderTable({

    inFile <- input$file1

    if (is.null(inFile))
      return(NULL)

    read.csv(inFile$datapath, header = input$header,
             sep = input$sep, quote = input$quote)
  })
```

In addition to creating a table so that our user may immediately see whether their file upload is successful (and we envision in a live scenario perhaps adding comments to the user interface to allow a user to spot a poor upload), we must also read the data in a format that will allow our markdown file to use that data. We read the data into a variable named `scores`. This is a reactive function that is always on the alert for changes in the relevant input variables (in this case, the uploading of a file).

```
scores <- reactive({
      inFile <- input$file1

      if (is.null(inFile))
        return(NULL)

      read.csv(inFile$datapath, header = input$header,
               sep = input$sep, quote = input$quote)
    })
```

The last piece of the server side is the most advanced. This is what controls the download file the user can access. The first part of this block controls the name of the report the user downloads. We named ours `quiz-report.pdf`, which only looks complex. That code gives us future flexibility should we change the type or nature of the report; it is easy to change our file type or name. Note that if you use RStudio's internal browser, the report downloads with a different name. Next, we call `normalizePath()` on our markdown file, which returns the file path of the current working directory with the filename in the argument appended. In particular, it returns this based on the operating system environment in which the code exists. This is important for two reasons. First, it allows portability of the code from Windows to Ubuntu (or any operating system). Second, we are about to change our working directory. Thus, a direct call to `report.Rmd` will no longer work.

It is important we change our working directory; we are about to create a new PDF file. Now, on a local machine we own and have administrative privileges to use, creating a new file is no trouble at all. However, our goal is to upload to the cloud, and we are not assured such privileges on all servers. Thus, we change our working directory to the temporary directory of the server. In R, `setwd(tempdir())` does two things. It both copies and returns the current working directory and then, after that, it changes the directory to the argument. Thus, `owd` is the old working directory. We set an `on.exit()` parameter so that, once `downloadHandler` is done, the working directory returns to what it ought to be, so that our Shiny site does not break.

We now copy our markdown file from its original location in the old working directory to our temp area, thereby allowing it to run and generate the PDF in that temp directory. We also set some environmental options for `knitr` (which creates the markdown PDF) that makes it possible to use the `tufte` format (named for Edward Tufte, http://rmarkdown.rstudio.com/tufte_handout_format.html). All that remains is to `render()` our markdown document and rename our file to what we wanted.

```
output$downloadReport <- downloadHandler(
  filename = function() {
    paste('quiz-report', sep = '.', 'pdf')
  },

  content = function(file) {
    src <- normalizePath('report.Rmd')

    owd <- setwd(tempdir())
    on.exit(setwd(owd))
    file.copy(src, 'report.Rmd', overwrite = TRUE)
```

```
    knitr::opts_chunk$set(tidy = FALSE, cache.extra = packageVersion('tufte'))
    options(htmltools.dir.version = FALSE)

    out <- render('report.Rmd')
    file.rename(out, file)
  }    )
```

What we have seen is that the Shiny server side needs to be able to read in the user's uploaded file with the appropriate input information, such as the existence of a header, and then is responsible for starting the `Rmarkdown` process. Admittedly, getting that process to work on any computer involves changing the working directory as well as some fiddling with layout and formatting options. We conclude this section by noting that a simple PDF would be easier to set up (although we admit to having a preference for the cleaner `tufte` layout. We turn our attention to the user interface.

ui.R

The user has a fairly clean view for this very simple Shiny application. In just 40 lines of code (and we could have removed several options, really), the user can upload the CSV file, view the rendered table from that file, and download the analysis of the file. As before, we first give the entire code and then describe each block in turn:

```
#Dynamic Reports and the Cloud User Interface
shinyUI(

  fluidPage(
    title = 'Score Control Panel',
    sidebarLayout(
      sidebarPanel(
        helpText("Upload a wide CSV file where
                 each column is 0 or 1 based on whether the student
                 got that question incorrect or correct."),

        fileInput('file1', 'Choose file to upload',
                  accept = c(
                    'text/csv',
                    'text/comma-separated-values',
                    'text/tab-separated-values',
                    'text/plain',
                    '.csv',
                    '.tsv'
                  )
        ),
        tags$hr(),
        checkboxInput('header', 'Header', TRUE),
        radioButtons('sep', 'Separator',
                     c(Comma=',',
                       Semicolon=';',
                       Tab='\t'),
                     ','),
        radioButtons('quote', 'Quote',
                     c(None='',
```

```
                    'Double Quote'='"',
                    'Single Quote'="'"),
                  '"'),
      tags$hr(),
      helpText("This is where helpertext goes"),
      downloadButton('downloadReport')
    ),
    mainPanel(
      tableOutput('contents')
    ))))
```

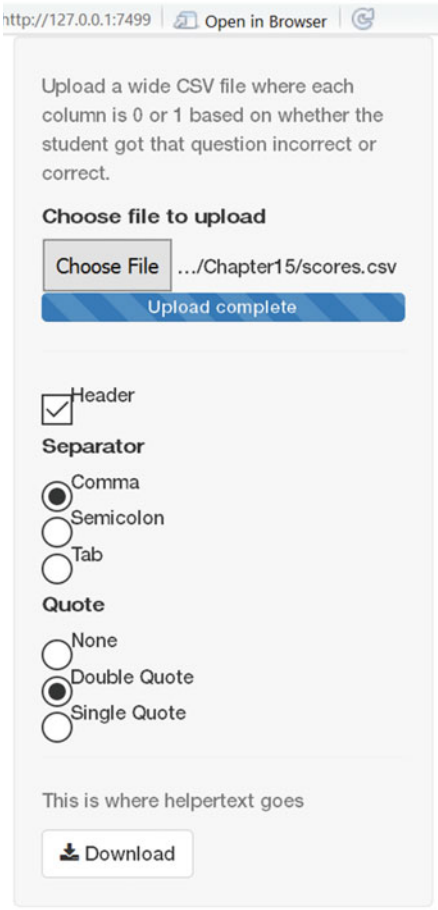
As usual for the user interface, we have `fluidPage()` along with `sidebarPanel()` and `mainPanel()`. It is inside the sidebar that the main events occur, and thus we look there first. The `fileInput()` function takes our input file. We can control the types of files accepted, and we limit ourselves to comma- or tab-separated values. We name the upload `file1`, recalling that on the server side we used `input$file1` to read in the file.

```
fileInput('file1', 'Choose file to upload',
          accept = c(
            'text/csv',
            'text/comma-separated-values',
            'text/tab-separated-values',
            'text/plain',
            '.csv',
            '.tsv'
          )
),
```

The only other line of major interest to us is the download area. It is not particularly complex to understand:

```
downloadButton('downloadReport')
```

Although there are helper text boxes to explain to the user, and we do recommend in real life setting up sample layouts so your users understand the acceptable inputs, we opted to keep this user interface as streamlined as possible. Before we turn our attention to the actual report file, we show the Shiny application in Figure 15-2 after uploading our `scores.csv` file (which is available on the Apress website for this book).



	Student	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10
1	S1	1	1	1	1	0	1	1	1	1	1
2	S2	1	0	1	1	0	1	0	1	0	1
3	S3	1	1	0	1	0	0	1	1	1	0
4	S4	1	1	1	1	1	1	0	1	1	1
5	S5	1	1	1	0	1	1	1	0	1	1
6	S6	0	1	1	1	1	0	0	1	1	1
7	S7	0	1	0	0	1	1	0	1	0	0
8	S8	0	0	0	1	0	0	0	1	0	1
9	S9	1	1	1	1	0	1	1	0	1	1
10	S10	1	1	1	1	1	1	1	1	1	1
11	S11	1	0	0	0	1	0	0	0	0	0
12	S12	1	0	1	0	1	1	0	1	1	1
13	S13	1	1	0	1	0	1	0	1	1	1
14	S14	0	1	1	1	1	0	1	1	1	1
15	S15	1	1	1	1	1	1	1	1	1	1
16	S16	1	0	1	1	1	1	1	1	1	0
17	S17	1	1	1	1	1	1	1	1	1	1
18	S18	1	1	1	0	1	0	0	0	1	0
19	S19	1	1	0	1	1	0	1	1	1	1
20	S20	0	0	1	0	0	1	0	0	1	0

Figure 15-2. *Shiny user interface, live on a locally hosted website*

report.Rmd

This report is about student scores per question for a quiz or other test. Our goal is to take the input shown in Figure 15-2 and convert it to actionable, summary information about question and test validity. Before reading past this first show of all the code, take a moment to compare and contrast the code to a stand-alone markdown file. There are no major differences. In fact, the only way to know that this file rendered from a Shiny application is a single line of code that reads `scores <- scores()`.

```
---
title: "Example Scoring Report"
subtitle: "Item-Level Analysis"
date: "`r Sys.Date()`"
output:
  tufte::tufte_handout: default
---
```

```
# Raw Data

Here is a sample of the data uploaded including the first few and last
rows, and first few and last columns:

```{r echo=FALSE, include = FALSE}
scores <- scores()
scores <- as.data.table(scores)
setnames(scores, 1, "Student")

```

```{r, echo = FALSE, results = 'asis'}
if (nrow(scores) > 6) {
 row.index <- c(1:3, (nrow(scores)-2):nrow(scores))
}

if (ncol(scores) > 6) {
 col.index <- c(1:3, (ncol(scores)-2):ncol(scores))
}

kable(scores[row.index, col.index, with = FALSE])
```

```{r, include = FALSE}
items <- names(scores)[-1]

scores[, SUM := rowSums(scores[, items, with = FALSE])]

## now calculate biserial correlations
## first melt data to be long
scores.long <- melt(scores, id.vars = c("Student", "SUM"))

## calculate biserial correlation, by item
## order from high to low
biserial.results <- scores.long[, .(
  r = round(biserial.cor(SUM, value, level = 2), 3),
  Correct = round(mean(value) * 100, 1)
  ), by = variable][order(r, decreasing = TRUE)]

alpha.results <- cronbach.alpha(scores[, !c("Student", "SUM"), with=FALSE])

rasch.results <- rasch(scores[,!c("Student", "SUM"), with=FALSE])

```

The test overall had `r ifelse(alpha.results$alpha > .6, "acceptable
reliability", "low reliability")` of
alpha = `r format(alpha.results$alpha, FALSE, 2, 2)`.^[Alpha ranges from 0 to 1, with one
indicating a perfectly reliable test.]
```

```
The graph shows the measurement error by level of
ability.^[Higher values indicate more measurement error, indicating the test is less
reliable at very low and very high ability levels (scores).]

```{r, echo = FALSE, fig.width = 5, fig.height = 4}

## The Standard Error of Measurement can be plotted by
vals <- plot(rasch.results, type = "IIC", items = 0, plot = FALSE)
plot(vals[, "z"], 1 / sqrt(vals[, "info"]),
     type = "l", lwd = 2, xlab = "Ability", ylab = "Standard Error",
     main = "Standard Error of Measurement")

```


Item Analysis

Results for individual items are shown in the following
table.^[*r* indicates the point biserial correlation of an item with the total score.
Correct indicates the percent of correct responses to a particular item. The items are
sorted from highest to lowest correlation.]

```{r, echo = FALSE, results = 'asis'}
kable(biserial.results)
```
```

We now break down and explain each block of this markdown file. This header is a trifle more advanced than the first. Notice that we use inline R code to set the date to the current date. Also, our output is no longer just a `pdf_document`. Instead, we use the default `tufte_handout` style (which is a PDF along with style information). We call that formally from its package (recall that `library(tufte)` was called back on the server side).

```
---
title: "Example Scoring Report"
subtitle: "Item Level Analysis"
date: "`r Sys.Date()`"
output:
  tufte::tufte_handout: default
---
```

Next up is the read into the markdown file of our score data. We are already familiar with `echo=FALSE`, which prevents the code from displaying. However, we also set `include=FALSE`, which prevents any output from the code from displaying. As noted earlier, we use the variable `scores()` from our Shiny server side, which is where we `read.csv()` our scores. Recall that there, on the Shiny server, we called that variable simply `scores`. However, in the `rmarkdown` file, we call it as a function (it does update dynamically, after all). Rather than continue calling a variable as a function, we rename it here with a local environment variable of the same name, `scores`:

```
```{r echo=FALSE, include = FALSE}
scores <- scores()
scores <- as.data.table(scores)
setnames(scores, 1, "Student")

```
```

Our next code chunk has `results='asis'` to prevent further processing of our `knitr` table (called `kable`). Additionally, this code will output the first three and the last two rows and columns into a small sample `kable`. It will do this only if there are more than six rows or columns.

```
```{r, echo = FALSE, results = 'asis'}
if (nrow(scores) > 6) {
 row.index <- c(1:3, (nrow(scores)-2):nrow(scores))
}

if (ncol(scores) > 6) {
 col.index <- c(1:3, (ncol(scores)-2):ncol(scores))
}

kable(scores[row.index, col.index, with = FALSE])
```
```

From here, we can calculate several measures of quiz or question reliability by using the `ltm` package. The point biserial correlation is essential; it uses the more familiar Pearson correlation adjusted for the fact that there is binary data for the quiz questions. It allows a comparison between students' performance on a particular question and their performance on the quiz overall (hence our comparison to the `rowSums`). By ordering this data from high to low, we may read off the quiz questions that have the lowest score. Those are the questions that may not be measuring the main theme of our quiz (admittedly, a vast oversimplification—psychometrics is a science well beyond the scope of this text). Additionally, we calculate both the Cronbach's alpha and Rasch model, which we use later. Again, we do not seek to include these results directly in our dynamic document; we can access them, though, for later use.

```
```{r, include = FALSE}
items <- names(scores)[-1]

scores[, SUM := rowSums(scores[, items, with = FALSE])]

## now calculate biserial correlations
## first melt data to be long
scores.long <- melt(scores, id.vars = c("Student", "SUM"))

## calculate biserial correlation, by item
## order from high to low
biserial.results <- scores.long[, .(
  r = round(biserial.cor(SUM, value, level = 2), 3),
  Correct = round(mean(value) * 100, 1)
  ), by = variable][order(r, decreasing = TRUE)]

alpha.results <- cronbach.alpha(scores[, !c("Student", "SUM"), with=FALSE])

rasch.results <- rasch(scores[,!c("Student", "SUM"), with=FALSE])

```
```

Not all code needs be inside the formal code blocks; code may be inline with the text of the document. This is helpful because we can set cutoff scores (in this case, semi-arbitrarily set at `0.6`) and generate

different text printed in the final report based on those scores. The tufte package allows for side column notes designated by ^[].

```
The test overall had `r ifelse(alpha.results$alpha > .6, "acceptable
reliability", "low reliability")` of
alpha = `r format(alpha.results$alpha, FALSE, 2, 2)`.^[Alpha ranges from 0 to 1, with one
indicating a perfectly reliable test.]

The graph shows the measurement error by level of
ability.^[Higher values indicate more measurement error, indicating the test is less
reliable at very low and very high ability levels (scores).]
```

Recall from our earlier discussion that figure size may be controlled in the code blocks. Here, we plot the Rasch model and see that high-performing and low-performing students are measured less precisely:

```
```{r, echo = FALSE, fig.width = 5, fig.height = 4}


## The Standard Error of Measurement can be plotted by
vals <- plot(rasch.results, type = "IIC", items = 0, plot = FALSE)
plot(vals[, "z"], 1 / sqrt(vals[, "info"]),
     type = "l", lwd = 2, xlab = "Ability", ylab = "Standard Error",
     main = "Standard Error of Measurement")

```
```

The code does end with one last kable, but we have seen that already and need not belabor the point. What we show now in Figure 15-3 is the final result of a download. Notice that all the suppressed code does not show up. Also, notice the sample of the data, since we know from Figure 15-2 that there were more than six rows or columns. Finally, notice the automatically numbered side comments that provide more detail. Those were built using the ^[] code inside the text area of the document. These make it very easy to provide detailed, yet not vital, insight.

Example Scoring Report

2016-09-18

Raw Data

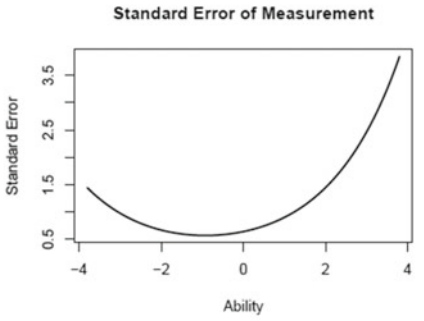
Here is a sample of the data uploaded including the first few and last rows, and first few and last columns:

Student	Q1	Q2	Q8	Q9	Q10
S1	1	1	1	1	1
S2	1	0	1	0	1
S3	1	1	1	1	0
S18	1	1	0	1	0
S19	1	1	1	1	1
S20	0	0	0	1	0

The test overall had acceptable reliability of alpha = 0.70.[1]
The graph shows the measurement error by level of ability.[2]

[1] Alpha ranges from 0 to 1, with one indicating a perfectly reliable test.

[2] Higher values indicate more measurement error, indicating the test is less reliable at very low and very high ability levels (scores).



Item Analysis

Results for individual items are shown in the following table.[3]

[3] *r* indicates the point biserial correlation of an item with the total score. *Correct* indicates the percent of correct responses to a particular item. The items are sorted from highest to lowest correlation.

variable	r	Correct
Q7	0.660	50

variable	r	Correct
Q9	0.649	80
Q10	0.603	70
Q2	0.558	70
Q4	0.558	70
Q3	0.513	70
Q1	0.452	75
Q8	0.405	75
Q6	0.402	65
Q5	0.272	65

Figure 15-3. *The PDF that downloads*

Uploading to the Cloud

Our last task is to get our files from a local machine to the cloud instance. Remember, in prior chapters, we have not only used PuTTY to install Shiny Server on our cloud, but also used WinSCP to move files up to our cloud. We start WinSCP, reconnect, and ensure that on the left, local side we are in our Shiny application folder, while on our right, cloud server side we are in /home/ubuntu. We recall the result of that process in Figure 15-4.

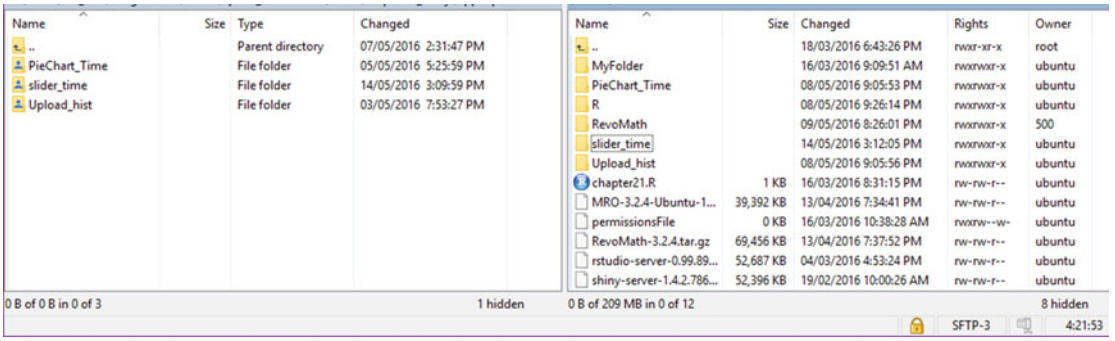


Figure 15-4. *WinSCP with PieChart_Time, slider_time, and Upload_hist Shiny applications all uploaded*

From here, we access PuTTY and run the following code from the command line:

```
sudo cp -R ~/Chapter15 /srv/shiny-server/
```

Remember, in an earlier chapter we adjusted our instance to allow access to certain ports from certain addresses. You may now verify that your application works by typing the following URL into any browser:

```
http://Your.Instance.IP.Address:3838/Chapter15/
```

Summary

In this, our last chapter, we built a dynamic document that can rapidly update based on new data. This update can even extend to graphics and inline `ifelse` statements, which allow the very narrative to morph based on new data. Additionally, we built a Shiny application that allows users with an Internet connection to upload a file to feed new data into such a report. Such reports can be download in many formats, although we chose PDF.

Throughout this text, we provided techniques for advanced data management, including connecting to various databases. Those techniques merge well with dynamic documents, and the ability to easily translate information in a data warehouse to actionable intelligence. Our final observation is that as data becomes more extensive, successful filtration and presentation of useful data become more vital skills in any field. Happy coding!

References

This chapter contains the full references for all resources cited throughout the rest of the book.

University Library, University of Illinois at Urbana-Champaign. "LibGuides: SPSS Tutorial: About This Tutorial." http://guides.library.illinois.edu/c.php?g=347869. Retrieved September 27, 2016.

Chambers, J. *Software for Data Analysis: Programming with R*. Springer, 2009.

Chang, W. "shinydashboard: Create Dashboards with 'Shiny'." R package version 0.5.1. https://CRAN.R-project.org/package=shinydashboard. 2015.

Chang, W., Cheng, J., Allaire, J., Xie, Y., and McPherson, J. "shiny: web application framework for R." R package version 0.11, 1. 2015.

Chheng, T. "RMongo: MongoDB Client for R." R package version 0.0.25. https://CRAN.R-project.org/package=RMongo. 2015.

Conway, J., Eddelbuettel, D., Nishiyama, T., Prayaga, S.K., and Tiffin, N. "RPostgreSQL: R interface to the PostgreSQL database system." R package version 0.4-1. https://CRAN.R-project.org/package=RPostgreSQL. 2015.

U.S. Census Bureau, American FactFinder. "Data Access and Dissemination Systems (DADS)." http://factfinder2.census.gov/faces/tableservices/jsf/pages/productview.xhtml?pid=ACS_10_5YR_B01003. Retrieved September 27, 2016.

Dragulescu, A. A. "xlsx: Read, write, format Excel 2007 and Excel 97/2000/XP/2003 files." R package version 0.5.7. https://CRAN.R-project.org/package=xlsx. 2014.

Dowle, M., Srinivasan A., Short T., and Lianoglou S. with contributions from R Saporta and E Antonyan. "data.table: Extension of Data.frame." R package version 1.9.6. https://CRAN.R-project.org/package=data.table. 2015.

Feinerer, I. and Hornik, K. "tm: Text Mining Package." R package version 0.6-2. https://CRAN.R-project.org/package=tm. 2015.

Fellows, I. "wordcloud: Word Clouds." R package version 2.5. https://CRAN.R-project.org/package=wordcloud. 2014.

Friedl, J. E. F. *Mastering Regular Expressions, Third Edition*. O'Reilly Media, 2006.

Harrell Jr, F. E. with contributions from Charles Dupont and many others. "Hmisc: Harrell Miscellaneous." R package version 3.17-1. https://CRAN.R-project.org/package=Hmisc. 2015.

Hester, J. "covr: Test coverage for packages." R package version 2.2.1. https://CRAN.R-project.org/package=covr. 2016.

Johnson, P. "devEMF: EMF Graphics Output Device." R package version 2.0. https://CRAN.R-project.org/package=devEMF. 2015.

Lang, D. T. and the CRAN Team. "XML: Tools for Parsing and Generating XML Within R and S-Plus." R package version 3.98-1.4. https://CRAN.R-project.org/package=XML. 2016.

Microsoft Corporation. "checkpoint: Install Packages from Snapshots on the Checkpoint Server for Reproducibility." R package version 0.3.16. https://github.com/RevolutionAnalytics/checkpoint. 2016.

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1

Ooms, J. "The jsonlite package: a practical and consistent mapping between JSON data and R objects." arXiv preprint arXiv:1403.2805. 2014.

Prikryl, M. *WinSCP—Free SFTP and SCP client for Windows*. 2007.

R Core Team. "foreign: Read Data Stored by Minitab, S, SAS, SPSS, Stata, Systat, Weka, dBase, ..." R package version 0.8-66. https://CRAN.R-project.org/package=foreign. 2015.

Rizopoulos, D. "ltm: An R package for latent variable modeling and item response theory analyses." *Journal of Statistical Software*, 17(5), 1–25. 2016.

R Special Interest Group on Databases (R-SIG-DB), Hadley Wickham, and Kirill Müller. "DBI: R Database Interface." R package version 0.5. https://CRAN.R-project.org/package=DBI. 2016.

RStudio Team. "RStudio: Integrated Development for R." www.rstudio.com. 2015.

Amazon Web Services. "Setting Up with Amazon EC2." http://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html. Retrieved October 2, 2016.

Schenk, C. MiKTEX project page. http//: miktex. org. 2009

Tatham, S. "PuTTY: A free SSH and Telnet client." www.chiark.greenend.org.uk/~sgtatham/putty/. 2016.

Urbanek, S. "rJava: Low-Level R to Java Interface." R package version 0.9-8. https://CRAN.R-project.org/package=rJava. 2016.

U. S. Census Bureau, American FactFinder. "B01003—"Total Population. All Counties within Illinois. 2006–2010 American Community Survey 5-Year Estimates." http://factfinder2.census.gov/faces/tableservices/jsf/pages/productview.xhtml?pid=ACS_10_5YR_B01003&prodType=table. 2010.

U. S. Census Bureau, American FactFinder. "B23006—Educational Attainment by Employment Status for the Population 25 To 64 Years. Universe: Population 25 to 64 years. All Counties within Illinois. 2006–2010 American Community Survey 5-Year Estimates." http://factfinder2.census.gov/faces/tableservices/jsf/pages/productview.xhtml?pid=ACS_10_5YR_B23006&prodType=table. 2010.

U. S. Census Bureau, American FactFinder.. "S1903—Median Income in the Past 12 Months (in 2010 Inflation-Adjusted Dollars): All Counties within Illinois. 2006–2010 American Community Survey 5-Year Estimates." http://factfinder2.census.gov/faces/tableservices/jsf/pages/productview.xhtml?pid=ACS_10_5YR_S1903&prodType=table. 2010.

van der Loo, M. "The stringdist package for approximate string matching." *The R Journal*, 6, pp. 111–122. http://CRAN.R-project.org/package=stringdist. 2014.

Venables, W., and Ripley, B. D. *S programming*. Springer, 2004.

*Wickham, H. "*Reshaping data with the reshape package." *Journal of Statistical Software, 21(12), pp. 1–20. 2007.*

Wickham, H. *ggplot2: Elegant Graphics for Data Analysis*. Springer, 2016.

Wickham, H. . "testthat: Get started with testing." *The R Journal, 3*(1), pp. 5–10.

Wickham, H. "evaluate: Parsing and Evaluation Tools that Provide More Details than the Default." R package version 0.9. https://CRAN.R-project.org/package=evaluate. 2016.

Wickham, H., and Chang, W. "devtools: Tools to Make Developing R Packages Easier." R package version 1.12.0. https://CRAN.R-project.org/package=devtools. 2016.

Wickham, H., Danenberg, P., and Eugster, M. "roxygen2: In-Source Documentation for R." R package version 5.0.1. https://CRAN.R-project.org/package=roxygen2. 2015.

Wickham, H. and Francois, R. "dplyr: A Grammar of Data Manipulation." R package version 0.4.3. https://CRAN.R-project.org/package=dplyr. 2015.

Wickham, H., Francois, R., and Müller, K. "tibble: Simple Data Frames." R package version 1.0. https://CRAN.R-project.org/package=tibble. 2016.

Wickham, H., James, D. A., and Falcon S. "RSQLite: SQLite Interface for R." R package version 1.0.0. https://CRAN.R-project.org/package=RSQLite. 2014.

Wiley, J. F. and Pace, L. *Beginning R: An Introduction to Statistical Programming, Second Edition*. Apress, 2015.

Xie, Y. *Dynamic Documents with R and knitr* . CRC Press, 2015.

Xie, Y. and Allaire, J. J. “tufte: Tufte’s Styles for R Markdown Documents.” R package version 0.2. https://CRAN.R-project.org/package=tufte. 2016.

YAML: The Official YAML Website. www.yaml.org. Retrieved October 2, 2016.

Zeileis, A. and Grothendieck, G. “zoo: S3 Infrastructure for Regular and Irregular Time Series.” Journal of Statistical Software *arXiv preprint math/0505527*.

Index

A

B

C

M. Wiley and J. F. Wiley, *Advanced R*, DOI 10.1007/978-1-4842-2077-1

D, E

F

P, Q

R

S

T

U

V

W

X, Y, Z