Second Edition

PROBABILITY *and* STATISTICS *with* R

María Dolores Ugarte
Ana F. Militino
Alan T. Arnholt

CRC Press
Taylor & Francis Group
A CHAPMAN & HALL BOOK

Second
Edition

PROBABILITY *and* STATISTICS *with* R

Second
Edition

PROBABILITY *and* STATISTICS *with* R

María Dolores Ugarte
Public University of Navarre
Pamplona, Navarre, Spain

Ana F. Militino
Public University of Navarre
Pamplona, Navarre, Spain

Alan T. Arnholt
Appalachian State University
Boone, North Carolina, USA

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business
A CHAPMAN & HALL BOOK

CRC Press
Taylor & Francis Group
6000 Broken Sound Parkway NW, Suite 300
Boca Raton, FL 33487-2742

Version Date: 20150227

International Standard Book Number-13: 978-1-4665-0440-0 (eBook - PDF)

Visit the Taylor & Francis Web site at
http://www.taylorandfrancis.com

and the CRC Press Web site at
http://www.crcpress.com

Contents

List of Figures

List of Tables

Preface to the Second Edition

Welcome to the second edition of *Probability and Statistics with R*! You are holding a text that will allow you to expand your practice of statistics into the current decade. The contributions that have been made to both packages and data since the first edition was published are truly astonishing. The number of contributed packages on CRAN has increased from around 1,000 to over 6,000, and this edition explores how some of those new packages can make analysis easier and more intuitive, as well as give more visually pleasing results in the case of graphs. The data associated with the book are available in the `PASWR2` package and have been augmented by numerous Internet sources for this edition. Furthermore, you will learn how to make use of the readily available data from numerous sites.

This text has improved over the first edition in terms of additions and clarifications of examples and problems, concepts, data, and functions. Most chapters of the book have several new examples and exercises that use the most modern functions and have problems to be solved with those functions to solidify understanding. Statistically, the concepts of the coverage probability of a confidence interval and model validation have been added to this version of the text. The text is supported at http://alanarnholt.github.io/PASWR2E-Book/ with solutions to odd exercises, and templates for homework assignments. A complete solutions manual is available from the publisher for text adopters. The package `PASWR2` written to support this text contains data sets and functions and is available on CRAN (http://cran.r-project.org/web/packages/PASWR2/index.html). The most recent updates to `PASWR2` can be found on GitHub at https://github.com/alanarnholt/PASWR2. Throughout, based on feedback of readers of the first edition, the prose has been expanded or rewritten as needed to make comprehension more certain.

Since R has become the undisputed language of choice for the majority of statistical practitioners, any references to its commercial counterpart, S-PLUS, have been removed. Additionally, the R code for calculations and graph creation has been highlighted for ease of reading and referenced with numbers as examples were in the first edition. Graphs have been created primarily with `ggplot2`, which has also moved to the fore as the most comprehensive graphics package with the greatest possibility of both customization and application.

It is our fervent hope that this edition of *Probability and Statistics with R* will serve you in your quest to discover truth in our world through the analysis of data. We welcome feedback and ideas for future editions and expect to continue both expansion and modification of the text as R itself continues to grow and change.

Acknowledgments

We gratefully acknowledge the invaluable help provided by Susie Arnholt, senior lecturer at Appalachaian State University. Her willingness to apply her expertise in LaTeX and knowledge of English grammar to the production of this text is appreciated beyond words. Many people were instrumental in improving the readability of this text; however, we are particularly appreciative of the contributions Tomás Goicoa, associate professor at the University of Navarre, made with respect to exercises, errata, and new ideas for the second edition.

Preface to the First Edition

The authors would like to thank their parents

Lola: Pedro and Loli

Ana: Carmelo and Juanita

Alan: Terry and Loretta

for their unflagging support and encouragement.

The Book

Probability and Statistics with R is a work born of the love of statistics and the advancements that have been made in the field as more powerful computers can be used to perform calculations and simulations that were only dreamed of by those who came before. The S language and its derivative, R, have made the practice of statistics available to anyone with the time and inclination to do so.

Teachers will enjoy the real-world examples and the thoroughly worked-out derivations. Those wanting to use this book as a reference work will appreciate the extensive treatments on data analysis using appropriate techniques, both parametric and nonparametric. Students who are visual learners will appreciate the detailed graphics and clear captions, while the hands-on learners will be pleased with the abundant problems and solutions. (A solutions manual should be available from Taylor & Francis.) It is our hope that practitioners of statistics at every level will welcome the features of this book and that it will become a valuable addition to their statistics libraries.

The Purpose

Our primary intention when we undertook this project was to introduce R as a teaching statistical package, rather than just a program for researchers. As much as possible, we have made a great effort to link the statistical contents with the procedures used by R to show consistency to undergraduate students. The reader who uses S-PLUS will also find this text useful, as S-PLUS commands are included with those for R in the vast majority of the examples.

This book is intended to be practical, readable, and clear. It gives the reader real-world examples of how S can be used to solve problems in every topic covered, including, but not limited to, general probability in both the univariate and multivariate cases, sampling distributions and point estimation, confidence intervals, hypothesis testing, experimental design, and regression. Most of the problems are taken from genuine data sets rather than created out of thin air. Next, it is unusually thorough in its treatment of virtually every topic, covering both the traditional methods to solve problems as well as many nonparametric techniques. Third, the figures used to explain difficult topics are exceptionally detailed.

Finally, the derivations of difficult equations are worked out thoroughly rather than being left as exercises. These features, and many others, will make this book beneficial to any reader interested in applying the S language to the world of statistics.

The Program

The S language includes both R and S-PLUS. "R can be regarded as an implementation of the S language which was developed at Bell Laboratories by Rick Becker, John Chambers, and Allan Wilks, and also forms the basis of the S-PLUS systems." (http://cran.r-project.org/doc/manuals/R-intro.html#Preface)

The current R is the result of a collaborative effort with contributions from all over the world. R was initially written by Robert Gentleman and Ross Ihaka of the Statistics Department of the University of Auckland. Since mid-1997 there has been a core group with write access to the R source (http://www.r-project.org/—click "Contributors"on the sidebar).

Not only is R an outstanding statistical package, but it is offered free of charge and can be downloaded from http://www.r-project.org/. The authors are greatly indebted to the giants of statistics and programming on whose shoulders we have stood to see what we will show the readers of this text.

The Content

The core of the material covered in this text has been used in undergraduate courses at the Public University of Navarre for the last ten years. It has been used to teach engineering (agricultural, industrial, and telecommunications) and economics majors. Some of the material in this book has also been used to teach graduate students studying agriculture, biology, engineering, and medicine.

The book starts with a brief introduction to S that includes syntax, structures, and functions. It is designed to provide an overview of how to use both R and S-PLUS so that even a neophyte will be able to solve the problems by the end of the chapter.

Chapter 2, entitled "Exploring Data,"covers important graphical and numerical descriptive methods. This chapter could be used to teach a first course in statistics.

The next three chapters deal with probability and random variables in a generally classical presentation that includes many examples and an extensive collection of problems to practice all that has been learned.

Chapter 6 presents some important statistics and their sampling distributions. Solving the exercises will give any reader confidence that the difficult topics covered in this chapter are understood.

The next four chapters encompass point estimation, confidence intervals, hypothesis testing, and a wide range of nonparametric methods including goodness-of-fit tests, categorical data analysis, nonparametric bootstrapping, and permutation tests.

Chapter 11 provides an introduction to experimental design using fixed and random effects models as well as the randomized block design and the two-factor factorial design.

The book ends with a chapter on simple and multiple regression analysis. The procedures from this chapter are used to solve three interesting case studies based on real data.

The Fonts

Knowing several typographical conventions will help the reader in understanding the material presented in this text. R code is displayed in a monospaced font with the > symbol in front of commands that are entered at the R prompt.

```
> x<-0.28354
> round(x,2)
  [1] 0.28
```

The same font is used for data sets and functions, though functions are followed by `()`. For example, the `PASWR` package but the `round()` function would be shown. Throughout the text, a ■ is found at the end of solutions to examples. In the index, page numbers in **BOLD** are where the primary occurrences of topics are found, while those in *ITALICS* indicate the pages where a problem about a topic or using a given data set can be located.

The Web

This text is supported at http://www1.appstate.edu/∼arnholta/PASWR on the Internet. The website has up-to-date errata, chapter scripts, and a copy of the `PASWR` package (which is also on CRAN) available for download.

Acknowledgments

We gratefully acknowledge the invaluable help provided by Susie Arnholt. Her willingness to apply her expertise in LaTeX and knowledge of English grammar to the production of this book is appreciated beyond words.

Several people were instrumental in improving the overall readability of this text. The recommendations made by Phil Spector, the Applications Manager and Software Consultant for the Statistical Computing Facility in the Department of Statistics at the University of California at Berkeley, who reviewed this text for Taylor & Francis, were used in improving much of the original R code as well as decreasing the inevitable typographical error rate. Tomás Goicoa, a member of the Spatial Statistics Research Group at the Public University of Navarre, was of great help in preparing and checking exercises. Celes Alexander, an Appalachian State University graduate student, graciously read the entire text and found several typos. Any remaining typos or errors are entirely the fault of the authors.

Thanks to our editor at Taylor & Francis, David Grubbs, for embracing and encouraging our project. Many thanks the Statistics and Operations Research Department at Public University of Navarre and to the Department of Mathematical Sciences at Appalachian State University for the support they gave us in our writing of this text.

The "You choose, you decide" initiative sponsored by Caja Navarra also provided funding for in-person collaborations. Thanks to the *Universidad Nacional de Educación a Distancia*, in particular the *Centro Asociado de Pamplona*, for allowing us to present this project under their auspices.

Special thanks to José Luis Iriarte, the former Vicerector of International Relations of the Public University of Navarre, and to T. Marvin Williamsen, the former Associate Vice Chancellor for International Programs at Appalachian State University. These men were instrumental in gaining funding and support for several in-person collaborations including a year-long visit at the Public University of Navarre for the third author and two multi-week visits for the first two authors to Appalachian State University.

Finally, to the geniuses of this age who first conceived of the idea of an excellent open source software for statistics and those who reared the idea to adulthood, our gratitude is immeasurable. May the lighthouse of your brilliance guide travelers on the ocean of statistics for decades to come. Thank you, R Core Team.

Chapter 1

What Is R?

1.1 Introduction to R

R is a language and an environment for statistical computing and graphics. In this book, R is used to store, manage, and manipulate data; to create graphical displays of data using different graphical systems; to analyze data using standard statistical procedures; and to perform simulations. In short, everything a student or scientist may want or need to do with data can be done with R, and this book uses R for everything that it covers. For the reader who wants his documents to update (graphs, tables, statistics, etc.) as the data changes, R integrates seamlessly with the typesetting program LaTeX via the Sweave and knitr packages. The `knitr` package also integrates R with markdown, an easy-to-use text markup language. The process of creating documents and solving problems that include all code and update as data changes is called reproducible analysis. Two excellent sources to learn more about the `knitr` package and reproducible analysis are *Dynamic Documents with R and knitr* (Xie, 2014) and *Reproducible Research with R and RStudio* (Gandrud, 2014), respectively.

The goals of this chapter are to provide sufficient detail to allow the reader to install R, along with an editor, on his operating system, to have the reader use R and the editor from the start in learning some of the basics of R syntax, to practice reading, writing, and accessing data, and to produce base graphs. Since R itself is a programming language, what one can do with R is truly amazing; however, this text will only address topics the authors have found useful in helping their students understand the material in a first calculus-based probability and statistics course. For a deeper introduction to the R language, the reader should consult *An Introduction to R*, which is available as a PDF file once R is installed. This can be done by clicking **Help**→**Manuals (in PDF)**→**An Introduction to R** or from http://cran.r-project.org/doc/manuals/R-intro.pdf. R can be used in batch mode as well as in interactive mode. This book only uses R in the interactive mode and has the reader type commands either at the R prompt or in an R script. It is the firm belief of the authors that typing commands as opposed to using drop-down menus leads to a better understanding of exactly what the reader/user is doing. While R runs on virtually any platform, the comments provided in the text are Microsoft Windows-specific unless otherwise noted.

1.2 Downloading and Installing R

Precompiled binary distributions of the base R system and contributed packages are available for Windows, MacOS X, and Linux operating systems. Source code is also available for the reader who wants to compile R from source or to run R on an operating system other

than Windows, MacOS X, or Linux. In spite of this, no attempt is made in this material to guide the reader through building R from source. The definitive reference for installing R, regardless of whether one is installing from source or installing from a precompiled binary distribution, is the *R Installation and Administration* manual, also available once R is installed. This manual is found by clicking on **Help**→**Manuals (in PDF)**→**R Installation and Administration** as well as online at http://cran.r-project.org/doc/manuals/R-admin.pdf. If problems are encountered installing a binary distribution, a quick answer to the particular difficulty will often be documented in Frequently Asked Questions (FAQs). To open the FAQs web page, click FAQs on the left menu at http://cran.r-project.org; then, choose the respective platform one is using (see Figure 1.1).

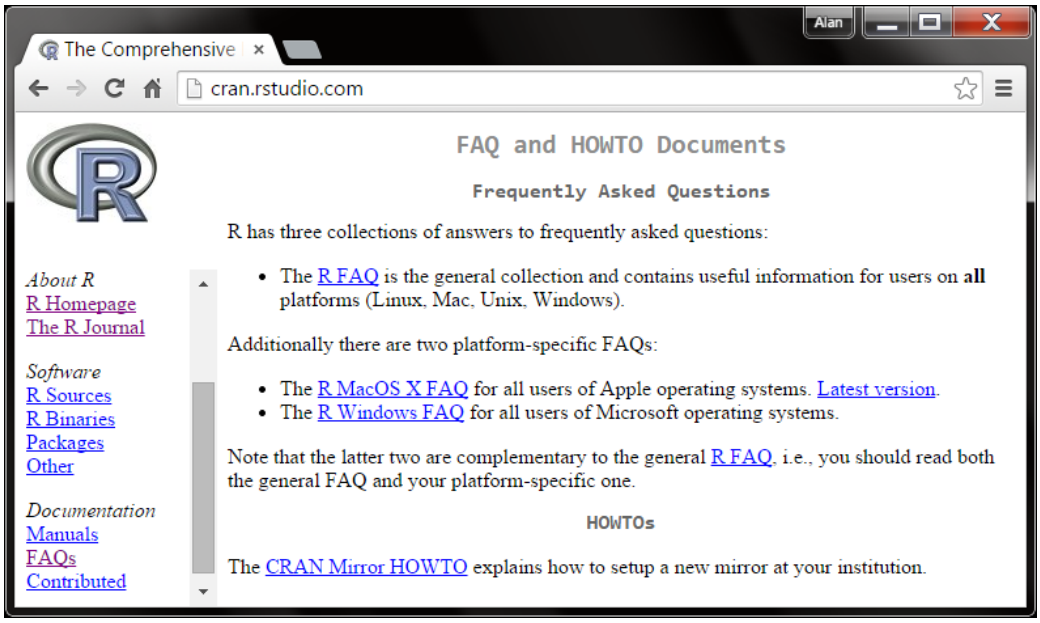

FIGURE 1.1: Frequently asked questions

1.2.1 Installing R under Windows

The home page for R is the website http://www.r-project.org. (See Figure 1.2 on the facing page.) By clicking on the **CRAN** link depicted on the left side of Figure 1.2 on the next page, one is presented with a selection of mirrors. If one selects "0-Cloud," your computer will automatically be redirected to a server close to your location. A window similar to Figure 1.3 on the facing page will open once a mirror location is selected. Clicking on the **Download R for Windows** link in Figure 1.3 on the next page opens a window similar to Figure 1.4 on the facing page, and clicking on the **base** link in Figure 1.4 on the next page produces Figure 1.5 on page 4.

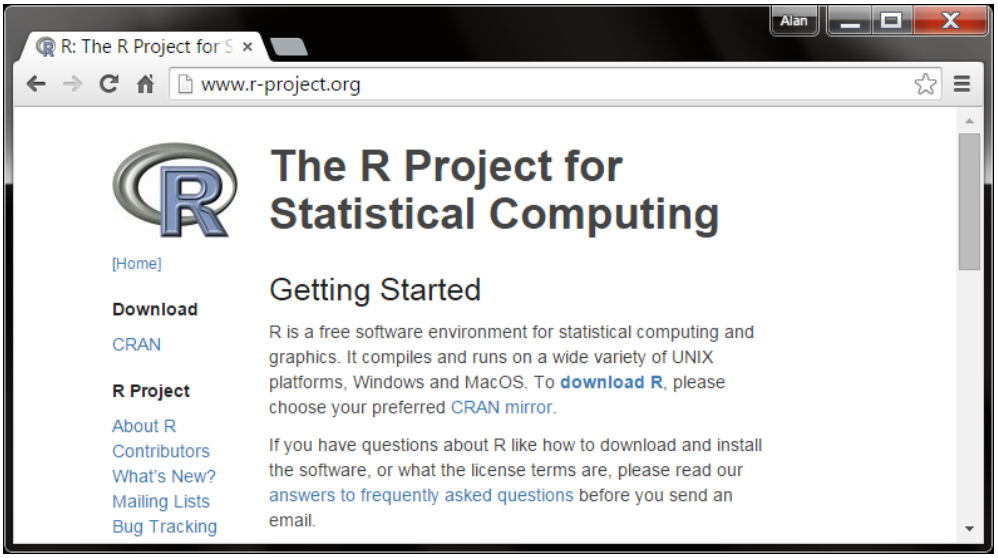

FIGURE 1.2: Home page for R

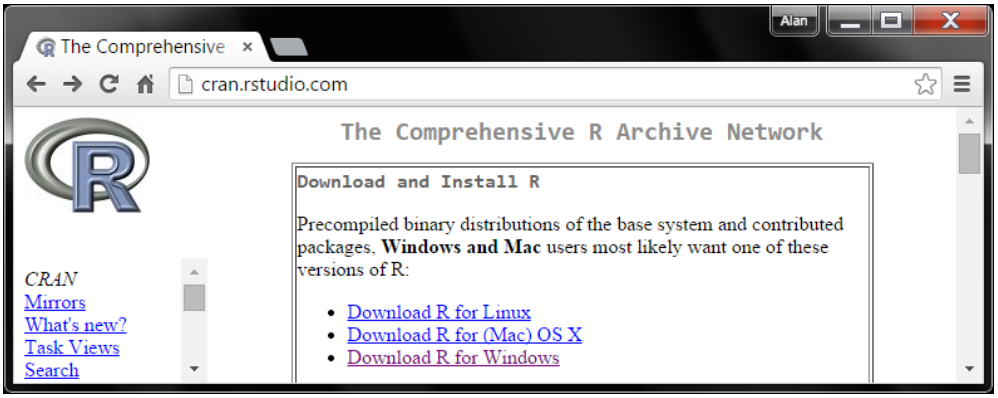

FIGURE 1.3: Download and install R

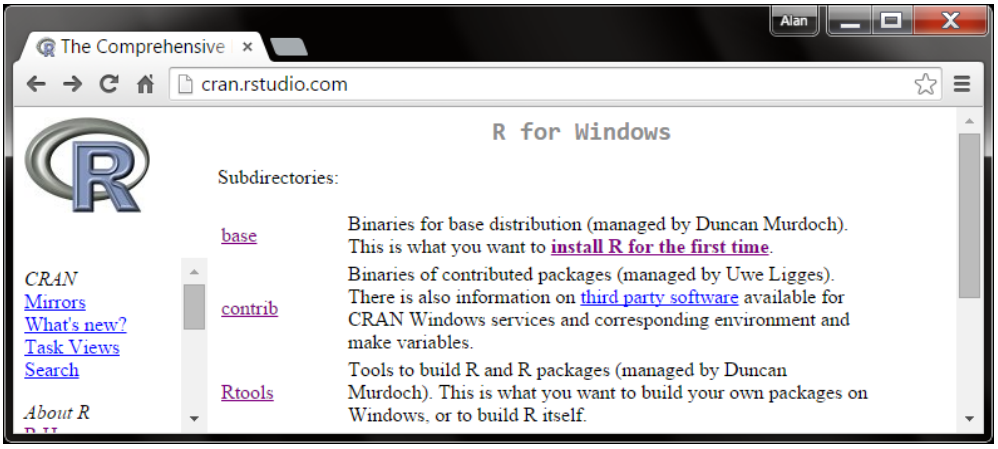

FIGURE 1.4: R for Windows

Click on the **Download R X.YY.x (R-3.2.0 as of June 2, 2015) for Windows** link, and the **R-3.2.0-win.exe** (or more current version, R-X.YY.x-win.exe) file will download to your computer. To install R, click on the downloaded **R-X.YY.x-win.exe** file. Choosing the default settings is most likely the easiest way to install R. For more information on the installation options, see the *R Installation and Administration* manual (http://cran.r-project.org/doc/manuals/R-admin.pdf). To install R for Mac OS X 10.6 (Snow Leopard) and higher, click on the **Download R for MacOS X** link as seen in Figure 1.3 on the previous page. Double click on the **R-3.2.0.pkg** (or more recent) file. For further information with other operating systems, refer to the *R Installation and Administration Manual*.

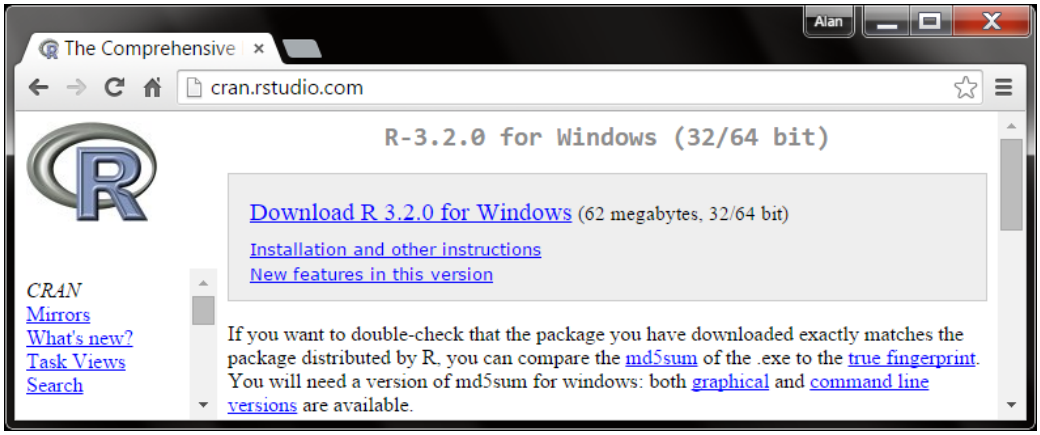

FIGURE 1.5: Download R for Windows link

1.2.2 Launching R

If R is installed on a Windows computer using the default options, the R icon will appear on the desktop as a shortcut icon. Double click the R icon, and R should open in a window resembling Figure 1.6 on the facing page. On Windows computers, R may also be launched by clicking the Windows **Start icon**→**All programs**→**R**→**R X.YY.x** (R 3.2.0 as of June 2, 2015). To launch R on a Mac, double click the R icon which, on a default installation, will be in the Applications folder. To launch R on a Linux platform, enter the command "R" at the system command line.

1.2.3 A First Look at R (Interactive Mode)

Standard mathematical operations (+ (addition), − (subtraction), / (division), * (multiplication), ˆ (raise to a power), etc.) can be used in a mathematical expression by typing in the R console at the R prompt (>). Once the user presses the Enter key, the result(s) of the requested mathematical operations appear below the code to the right of square brackets enclosing a number indicating the index of the answer for the value immediately to the right of the enclosed number.

For example, R Code 1.1 on the next page requests that twenty randomly generated values from a continuous uniform distribution with a minimum value of 0 and a maximum

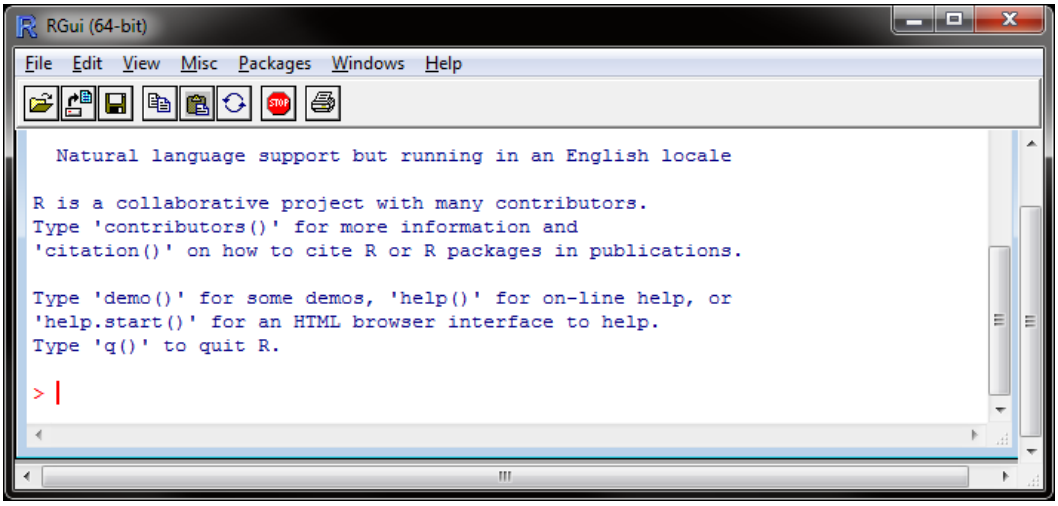

FIGURE 1.6: R Console running in Windows

value of 1 be stored in the R object `ruv` using the assignment operator (`<-`). Assignment may also be made with the equal sign (=). There are instances when the use of an = may lead to confusion, so the authors use the traditional assignment operator (`<-`) as a general rule. The value immediately to the right of `[1]`, 0.0694, is the first value; and the value immediately to the right of `[19]`, 0.6657, is the nineteenth value. When illustrating pedagogical concepts, the user will often want to generate the same set of "random" numbers at a later date. To reproduce the same set of "random" numbers, one uses the `set.seed()` function. The `set.seed()` function puts the random number generator in a reproducible state. Throughout the text, the width of printed output is set to a maximum number of 70 columns on a line using `options(width = 70)`. Text to the right of a `#` sign is not evaluated by R and is used to make comments about code.

R Code 1.1

```
> options(width = 70)  # set width of console output
> set.seed(12)  # set seed to make results reproducible
> ruv <- runif(n = 20, min = 0, max = 1)
> round(ruv, 4)  # round answers to 4 decimal places

 [1] 0.0694 0.8178 0.9426 0.2694 0.1693 0.0339 0.1788 0.6417 0.0229
[10] 0.0083 0.3927 0.8139 0.3762 0.3808 0.2649 0.4393 0.4576 0.5407
[19] 0.6657 0.1127
```

To use R as a calculator to compute $(7 \times 3) + 12 \div 2 - 7^2 + \sqrt{4}$, enter

```
> (7 * 3) + 12/2 - 7^2 + sqrt(4)

[1] -20
```

The answer to the previous computation is `-20`, printed to the right of `[1]`, which indicates the answer starts at the first element of the vector. Common functions such as `log()`, `log10()`, `exp()`, `sin()`, `cos()`, `tan()`, and `sqrt()` (square root) are all recognized by R. For a quick reference to commonly used R functions, see Table A.1 on page 903. If the user omits a comma or a parenthesis, or any other type of syntax error occurs when typing at

the command line, a + sign will appear to indicate that the command is incomplete. If the user does not recognize a typographical error, an easy technique to get rid of the + sign is to press the `Esc` key for Windows and Mac users working in the console. To interrupt R in Linux or in a terminal on a Mac, type `control-C`.

1.3 Vectors

R has many types of data structures, and one of the more important is the vector. All of the elements of a vector must have the same mode, that is, data type. One might have a vector of all numeric values, all character values, or all logical values, but not a mixture of data types. If a vector is created with mixed modes, the individual elements are all forced to a single mode such as character or numeric. Single numbers, often called scalars, do not exist as scalars in R. A single number in R is simply a vector with one element.

This important paradigm will be instrumental in understanding how R recycles values to work with unequal-length vectors. For example, how do vectors with more than one element interact with vectors containing a single element (scalars)? Mathematical operations may be applied to two vectors provided the vectors are of the same length. When the two vectors are not of the same length, R recycles the values of the shorter vector until the length of the shorter vector matches that of the longer vector.

Consider R Code 1.2 where `x` is a single element vector (scalar) and `y` and `z` are vectors of lengths 3 and 4, respectively. One way to construct a vector is with the `c()` function, which combines values into either a vector or a list. When `x` (a scalar) is added to `y` (a vector), the value in the scalar is added to each element in the vector `y`. What happens is that `x` is matched in length to `y` by recycling its single value. Then, the elongated `x` is added to `y`, element-wise. When a vector of a single element (scalar) is added to a vector of more than one element, no warning appears; however, when applying mathematical operations to two vectors of unequal length (neither of which is a scalar), a warning message will appear telling the user that the lengths of the vectors are not the same. What the warning message does not tell the user is that recycling has occurred with the shorter vector to grow it to the length of the larger vector.

R Code 1.2

```
> x <- 5                # vector of length one
> y <- c(7, 3, 5)       # combining the values into y
> z <- c(2, 4, 6, 8)    # combining the values into z
> x + y                 # adding x and y

[1] 12  8 10

> c(5, 5, 5) + y        # the shorter vector is recycled 3 times

[1] 12  8 10

> y + z                 # adding y and z

Warning in y + z:  longer object length is not a multiple of shorter object
length

[1]  9  7 11 15
```

```
> c(y, 7) + z          # values of y recycled to length of z

[1]  9  7 11 15
```

When `x` is added to `y`, the value in `x` (5) is recycled until the length of `x` is equal to the length of `y`. Then, the elements of the newly augmented `x` are added to the elements of `y`, one value per index location. When the vector `y` is added to `z`, the length of `y` must first be increased to equal the length of `z`. Since the length of `y` is three, it requires only one value, the first element in `y` (7), to be recycled to obtain a length of four for `y`.

Asking R if the values of `x` are less than the values of `z` returns a logical value (`TRUE/FALSE`) for each comparison. Logical values are not enclosed in quotes, and a `TRUE` is stored internally as a `1` and a `FALSE` as a `0`. The logical operators are `>` for greater than, `<` for less than, `<=` for less than or equal to, `>=` for greater than or equal to, `==` for exact equality, `!=` for exact inequality, `&` for vector intersection, `|` for vector union, `&&` for scalar intersection, and `||` for scalar union. Although R does not actually work with scalar quantities, different types of Boolean operators exist for vectors and scalars. To determine the storage mode of an object, one can use the function `typeof()`. R Code 1.3 compares the elements in `x` to see if they are less than the elements in `z`, and the function `typeof()` is used to verify that the vector `LogVec` is a logical vector.

R Code 1.3

```
> LogVec <- (x < z)  # logical vector
> LogVec             # 5 < 2, 5 < 4, 5 < 6, 5 < 8

[1] FALSE FALSE  TRUE  TRUE

> typeof(LogVec)     # determine how LogVec is stored internally

[1] "logical"
```

Two logical vectors `X` and `Y`, which are different from the lower case `x` and `y` created earlier, are used to illustrate vector intersection and vector union in R Code 1.4. Both ordered elements must be true for the resulting intersection to be `TRUE`.

R Code 1.4

```
> X <- c(FALSE, TRUE, FALSE)
> Y <- c(FALSE, TRUE, TRUE)
> X & Y  # Boolean X intersection Y

[1] FALSE  TRUE FALSE

> X | Y  # Boolean X union Y

[1] FALSE  TRUE  TRUE

> X == Y  # Boolean EQUALITY

[1]  TRUE  TRUE FALSE

> X != Y  # Boolean INEQUALITY

[1] FALSE FALSE  TRUE
```

When using scalar intersection (`&&`) or scalar union (`||`), only the first elements of each vector are compared.

```
> X && Y  # only looks at first element of each vector (intersection)

[1] FALSE

> X || Y  # only looks at first element of each vector (union)

[1] FALSE
```

Elements other than the first can be compared by specifying the index of the desired element in square brackets. For example, to compare the third element of `Y` with the second element of `X` with scalar intersection one would use `Y[3] && X[2]`. Examples comparing elements other than the first elements from two vectors are illustrated in R Code 1.5.

R Code 1.5

```
> Y[3] && X[2]  # compares 3 element of Y to 2 element of X

[1] TRUE

> X[2] && Y[2]  # compares second element of each vector (intersection)

[1] TRUE

> X[2] && Y[3]  # compares element 2 of X and element 3 of Y (intersection)

[1] TRUE

> X[3] || Y[3]  # compares third element of both vectors (union)

[1] TRUE
```

Character elements surrounded by quotes stored in a vector are considered to be of mode character. Objects of different modes stored together are automatically converted to the simplest mode to represent the information. The order of complexity starting with the simplest mode is usually: logical, integer, numeric/double, complex, character, and list. R can coerce objects to have different modes using one of `as.logical()`, `as.integer()`, `as.numeric()`, `as.double()`, `as.complex()`, `as.character()`, or `as.list()` functions. R has two names for its floating point vectors, double and numeric. In R Code 1.6, a logical vector is combined with an integer vector, and R subsequently coerces the resulting vector `IntVec` into an integer vector. The objects used in R Code 1.6 were created in previous R Code chunks starting with R Code 1.2 on page 6.

R Code 1.6

```
> typeof(z)               # determine how z is stored internally

[1] "double"

> z1 <- as.integer(z)     # coerce z from double to integer
> z1

[1] 2 4 6 8
```

```
> typeof(z1)              # determine how z is stored internally

[1] "integer"

> IntVec <- c(LogVec, z1) # integer  vector
> IntVec

[1] 0 0 1 1 2 4 6 8

> typeof(IntVec)          # determine how IntVec is stored internally

[1] "integer"
```

When a logical vector (`LogVec`) is combined with a numeric vector (`z`) and stored in the object `NumVec`, `NumVec` is stored internally as a numeric vector.

```
> NumVec <- c(LogVec, z)  # numeric vector
> NumVec

[1] 0 0 1 1 2 4 6 8

> typeof(NumVec)  # determine how NumVec is stored internally

[1] "double"
```

When a complex number is added to a numeric vector (`NumVec`) and stored in the object `ComVec`, `ComVec` is stored internally as a complex vector.

```
> ComVec <- c(NumVec, 0+0i)  # complex vector
> ComVec

[1] 0+0i 0+0i 1+0i 1+0i 2+0i 4+0i 6+0i 8+0i 0+0i

> typeof(ComVec)  # determine how ComVec is stored internally

[1] "complex"
```

When a character string (`"dog"`) is added to a complex vector (`ComVec`) and stored in the object `ChrVec`, `ChrVec` is stored internally as a character vector.

```
> ChrVec <- c(ComVec, "dog")  # a character vector
> ChrVec

 [1] "0+0i" "0+0i" "1+0i" "1+0i" "2+0i" "4+0i" "6+0i" "8+0i" "0+0i" "dog"

> typeof(ChrVec)  # determine how ChrVec is stored internally

[1] "character"
```

When a list is combined with a character vector (`ChrVec`) and stored in the object `Lst`, `Lst` is stored internally as a list.

```
> Lst <- c(ChrVec, list(x = 4))  # a list
> typeof(Lst)  # determine how Lst is stored internally

[1] "list"
```

To this point, names for variables have not appeared to have many restrictions, which is the case. Names for variables, or more appropriately for objects, can be constructed from letters, digits, and the period symbol, with the caveat that the name of the object cannot start with a digit or a period followed by a digit. It is also permissible to use the underscore between characters such as `this_one`. Finally, R is case sensitive, so the variables `ABC`, `Abc`, and `aBc` are all considered different.

1.3.1 Naming Cautions

Though R has flexible naming conventions, there are certain names one should not use. One should exercise care not to use names of functions such as `c`, `C`, `q`, `t`, `T`, `mean`, `median`, and so on, since doing so will change the meaning of said functions in the current session and may result in unexpected consequences if any code calls a function that has been overwritten. Furthermore, there are reserved words that should not be used in object names. The reserved words are `if`, `else`, `repeat`, `while`, `function`, `for`, `in`, `next`, `break`, `TRUE`, `FALSE`, `NULL`, `Inf`, `NaN`, `NA`, `NA_integer_`, `NA_real_`, `NA_complex_`, and `NA_character_`.

1.3.2 Vector Indexing

Indices for R vectors start at 1, unlike indices in C and C++, which start at 0. Individual elements of a vector are accessed with `[ ]` (square brackets). R uses `NA` to represent missing values, `NaN` to represent "not a number," and `Inf` and `-Inf` to represent very large and small numbers, respectively. R has a few built-in constants such as: `LETTERS`, the 26 upper-case letters of the Roman alphabet; `letters`, the 26 lower-case letters of the Roman alphabet; and `pi`, the ratio of the circumference of a circle to its diameter. To read about other constants, type `help("Constants")` at the R prompt. R Code 1.7 returns `NaN`, `-Inf`, and `Inf` values.

R Code 1.7

```
> NV <- c(-4, 0, 2, 4, 6)  # numeric vector
> NV/NV  # 0/0 is not a number (NaN)

[1]   1 NaN   1   1   1

> sqrt(NV)  # cannot take square root of -4

Warning in sqrt(NV): NaNs produced

[1]      NaN 0.000000 1.414214 2.000000 2.449490

> NV^9999  # very large and small numbers use -Inf and Inf

[1] -Inf    0  Inf  Inf  Inf
```

Elements can be omitted from a vector by using a vector of negative indices inside the square brackets. Positive indices extract the indexed values of the vector. It is not possible to index values of a vector using both positive and negative indices in a single call.

```
> NV[-1]  # omit first element of NV

[1] 0 2 4 6

> NV[c(1, 3)]  # extract first and third element of NV
```

```
[1] -4  2

> CV <- LETTERS[c(1, 2, 3, 4)]  # first four upper case letters of CV
> CV

[1] "A" "B" "C" "D"

> CV[c(2, 3)]  # Extract second and third elements of CV

[1] "B" "C"

> LV <- c(TRUE, FALSE, TRUE, TRUE)
> LV

[1]  TRUE FALSE  TRUE  TRUE

> LV[-2]  # omit second element of LV

[1] TRUE TRUE TRUE

> LV[-c(1, 3)]  # omit first and third elements of LV

[1] FALSE  TRUE
```

1.3.3 Generating Vector Sequences and Repeating Vector Constants

The `:` operator and the `seq()` function can be used to generate useful sequences. The `:` operator generates regular sequences from a starting value to an ending value of the form `from:to`. In the previous section, the first four capital letters of the alphabet were specified by typing `LETTERS[c(1, 2, 3, 4)]`. An equivalent solution requiring less typing is `LETTERS[1:4]`. To create patterned, nonconsecutive or non-integer values, one can use the `seq()` function, which allows one to specify an increment using the `by=` argument and the desired length of the sequence using the `length.out=` argument. Consider the sequences shown in R Code 1.8.

R Code 1.8

```
> 24:20  # values 24, 23, 22, 21, and 20

[1] 24 23 22 21 20

> letters[24:20]  # 24th through 20th lowercase letters

[1] "x" "w" "v" "u" "t"

> seq(from = 5, to = 25, by = 5)

[1]  5 10 15 20 25

> seq(from = 5, by = 5, length.out = 5)

[1]  5 10 15 20 25

> seq(from = 23, to = 22, length.out = 5)

[1] 23.00 22.75 22.50 22.25 22.00
```

The function `rep()` allows the user to create vectors with repeated constants stored in a vector passed to the argument `x=` by judicious use of the function's arguments `times=`, `each=`, and `length.out=`. R Code 1.9 shows examples of the function `rep()`.

R Code 1.9

```
> rep(x = 5, times = 10)

 [1] 5 5 5 5 5 5 5 5 5 5

> rep(x = c(TRUE, FALSE), times = 10)

 [1]  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE
[12] FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE  TRUE FALSE

> rep(x = letters[5:8], each = 2)

[1] "e" "e" "f" "f" "g" "g" "h" "h"

> rep(x = 13:11, times = 1:3)

[1] 13 12 12 11 11 11

> rep(x = 1:5, length.out = 20)

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

1.3.4 Filtering Vectors

Filtering allows the user to extract elements of a vector that satisfy certain conditions. Consider R Code 1.10, which creates the vector `FEV`, returns a vector of Boolean values for the expression `FEV*FEV > 3`, and extracts the elements of `FEV` where `FEV*FEV > 3` is true.

R Code 1.10

```
> FEV <- rep(x = 3:1, times = 1:3)
> FEV

[1] 3 2 2 1 1 1

> FEV * FEV

[1] 9 4 4 1 1 1

> FEV * FEV > 3

[1]  TRUE  TRUE  TRUE FALSE FALSE FALSE

> FEV[FEV * FEV > 3]

[1] 3 2 2

> # An equivalent approach using subset
> subset(x = FEV, subset = FEV * FEV > 3)
```

```
[1] 3 2 2

> # Instead of the actual values we may want the indices
> # where these values occur
> which(FEV * FEV > 3)

[1] 1 2 3
```

The function `subset()` returns subsets for vectors, matrices, and data frames. There are two basic arguments: `x=`, the object to be subsetted, and `subset=`, a logical expression indicating elements or rows to keep that need to be supplied regardless of the type of object (vector, matrix, data frame) one is using. When working with matrices or data frames, one may also use the argument `select=`, which requires an expression indicating the columns to select from the matrix or data frame.

1.4 Mode and Class of an Object

All objects in R have a mode and class that describe how the object is stored. The function `mode()` returns the mode of an object, and the function `class()` returns the class of an object. Many functions in R will behave differently depending on the class of their arguments. Functions that behave in this fashion are known as generic functions. Two common generic functions are `print()` and `summary()`. Although the only type of data structure discussed thus far has been the vector, which was described as a variable and subsequently called an object, everything in R is actually an object. Stored results from analyses, matrices, arrays, vectors, and so on, are all considered objects in R. Observe the mode and class of the objects created in R Code 1.11.

R Code 1.11

```
> Num <- c(1, pi, 5)
> Log <- c(TRUE, FALSE, TRUE)
> Chr <- c("a", "character", "vector")
> mode(Num)

[1] "numeric"

> class(Num)

[1] "numeric"

> mode(Log)

[1] "logical"

> class(Log)

[1] "logical"

> mode(Chr)

[1] "character"

> class(Chr)

[1] "character"
```

It is not always the case that the mode and class of an object coincide. The mode of a linear model object is stored as a list; however, the class of a linear model object is `lm`. Consider R Code 1.12, which stores the results from regressing `Y2` onto `x2` in an object named `model`. R uses the tilde (`~`) to separate the left and right sides in a model formula. A formula such as `Y ~ x1 + x2` is read `Y` is modeled by `x1` and `x2`.

R Code 1.12

```
> x2 <- 1:5
> Y2 <- x2 + rnorm(n = 5, mean = 0, sd = 0.5)  # add normal errors
> model <- lm(Y2 ~ x2)  # Regressing Y2 onto x2
> mode(model)  # model has mode list

[1] "list"

> class(model)  # model has class lm

[1] "lm"
```

1.5 Getting Help

R has extensive online help as well as HTML files one can access with a web browser. To access the HTML help, type `help.start()` at the R command prompt. The HTML version of the help system has a very useful "Search Engine & Keywords." To open the help file of a particular function, say `median()`, one might type `?median` or `help(median)` at the R command prompt. Most help files contain useful examples of what they are describing. To run the examples in a help file without copying and pasting the code from the file, one can use the function `example()`. R Code 1.13 runs the examples given in the median's help file.

R Code 1.13

```
> example(median)


median> median(1:4)                 # = 2.5 [even number]
[1] 2.5

median> median(c(1:3, 100, 1000))  # = 3 [odd, robust]
[1] 3
```

Many functions will have a long list of arguments. To see the arguments of a function, use the `args()` function. To see the arguments of the `lm()` function, type `args(lm)` at the R prompt.

```
> args(lm)

function (formula, data, subset, weights, na.action, method = "qr",
    model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
    contrasts = NULL, offset, ...)
NULL
```

Another way to view the arguments of a function, although fleeting, is to use tab completion. After entering a function and its opening parenthesis, press the tab key, and the arguments for the function appear above the cursor. The arguments disappear as soon as a value is specified in the argument list, yet they may be recalled by pressing the tab key again, provided the closing parenthesis has not been typed. Other ways to view arguments will be discussed later in conjunction with external editors.

In addition to HTML help, the reader may consult any one of the eight (on Windows-based machines) R manuals by selecting **Help→Manuals (in PDF)**. Six of the eight manuals are available under **Help→HTML help** under the *Manuals* heading. If the help files have not provided an answer to your query, it is likely your question has been discussed in the past on the R newsgroup. To view one of the searchable archives from the newsgroup, inside of R, click **Help→R Project home page** (or manually open the URL www.r-project.org), then click on the **Mailing Lists** link along the left side of the page. Move to the R-help section, and click **web-interface**. For Windows users, inside R, one can select **Help→search.r-project.org** or manually open the website http://search.r-project.org. Stack Overflow (http://stackoverflow.com) is a searchable site that addresses many different programming issues. Use the "r" tag to find or post an R related programming question on Stack Overflow.

If you still do not find the answer to your question, as a last resort, you might consider posting a well-constructed question to the R help list. Before posting to the R help list, one should read and follow the posting guide at http://www.r-project.org/posting-guide.html. Failure to read and follow the posting guide often results in online chastisement for failure to do so when a question does not conform to the posting guidelines.

1.6 External Editors

External editors provide many advantages for command line programming, including syntax highlighting, parentheses matching, and commenting and un-commenting blocks of code. There are many editors the user can select, and the comments about the various editors are based on the authors' experiences teaching the material in this book. Most readers of a first calculus-based probability and statistics course have not been interested in investing the time needed to master Emacs Speaks Statistics (http://stat.ethz.ch/ESS/), an extremely powerful scripting editor capable of interacting with various statistics analysis programs such as R, SAS, S-Plus, and Stata; however, Vincent Goulet has collected and distributes a modified version of GNU Emacs that includes a few add-ons that are very easy to install for both Windows and MAC users (http://vgoulet.act.ulaval.ca/en/emacs/). One of the authors has used Eclipse (http://www.eclipse.org/) with the StatET plug-in (http://www.walware.de/goto/statet) to teach the material in this book to computer science majors who have already had exposure to Eclipse in their programming courses. Unfortunately, those students without previous exposure to Eclipse struggled, even with detailed directions on how to set up and use that editor, which detracted from the primary focus of teaching basic probability and statistics concepts. All of the authors have used Tinn-R (http://www.sciviews.org/Tinn-R/) as an editor to teach the material in this book. Regrettably, students using operating systems other than Windows have been placed at a slight disadvantage, as Tinn-R only runs on Windows. Another Windows-only editor is the RWinEdt package (http://cran.r-project.org/web/packages/RWinEdt/index.html) that is used in conjunction with the LaTeX editor WinEdt (http://www.winedt.com/), a share-

ware editor. The RWinEdt package provides a nice interface to R for Windows users who are already familiar with the LaTeX editor WinEdt (http://www.winedt.com/). All of the aforementioned editors have their followers and ardent supporters. For additional editors, point your browser to http://www.sciviews.org/_rgui/projects/Editors.html.

What editor do the authors recommend you use? The answer depends largely on your personal preferences. For teaching the material in this book to students using various operating systems, the authors' students have had the best experience with RStudio (http://www.rstudio.com/) in terms of configuration, installation, and general ease of use. While an editor is not required to run R or to use the material in this book, using one is highly recommended. What follows are a few pointers for using RStudio.

1.7 RStudio

RStudio is an integrated development environment (IDE) for using and programming R. The desktop version of RStudio runs on all major platforms (Windows, Mac, and Linux). Like R, it is an open-source project, which means that it can be downloaded and installed from the Internet for free. Before installing RStudio, one should have a relatively recent installation of R. For Windows and Mac OS X users, one simply downloads the self-installing binary from http://www.rstudio.org/download/desktop. Figure 1.7 shows a screenshot of how the desktop version of RStudio appears in Windows.

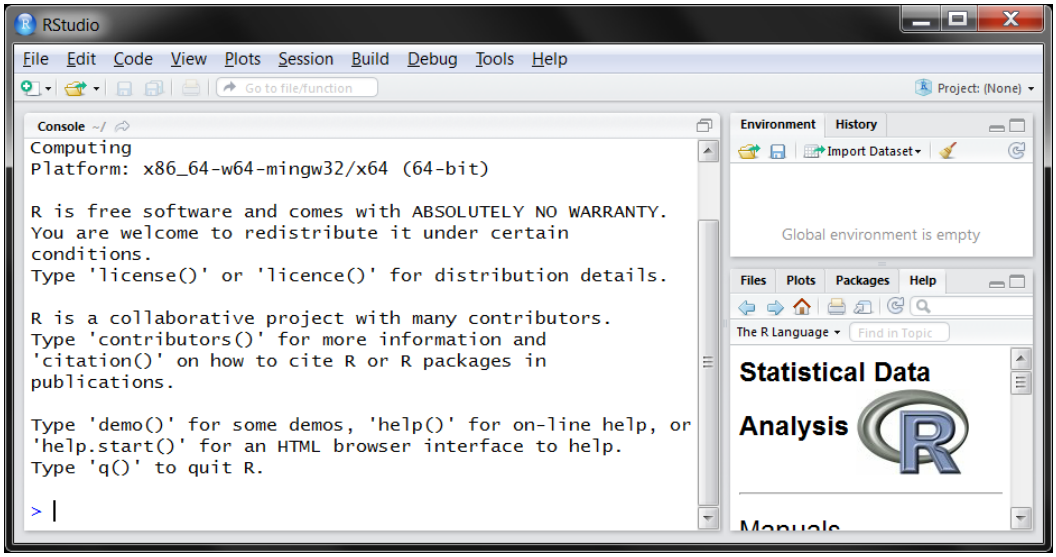

FIGURE 1.7: Screenshot of RStudio desktop

In Figure 1.7, there are three main windows: the *Console*, an *Environment* browser (currently empty), and the *Help* browser. The *Environment* browser and the *Help* browser are part of notebooks that contain other components (History, Files, Plots, and Packages). The source code editor is not open in the screenshot, as no files have been opened to view or

edit. To create a new R script, choose **File**→**New**→**R Script**. Type the code in the editor; then, select **File**→**Save** and give the script a name. The extension `.R` will automatically be appended to the file name, and the file will be saved in R's default working directory. For Windows machines, the default working directory is the `Documents` folder. For Linux and Mac OS X users, the directory from which R is launched is the working directory. To verify the working directory, one can enter the command `getwd()` at the R prompt.

Notice that the path is specified using forward slashes even for a Windows machine. To use the backwards slash with Windows, one must enter two backward slashes. This is because the backward slash is used in R to start an escape sequence inside character constants. As you work on different projects, or possibly from different chapters of this book, it is a good idea to save your work to named folders. For example, suppose you want to save the work for this chapter to a folder named `Chap1`. First, create the folder; then, change the working directory to where the `Chap1` folder resides using the `setwd()` command. If you are using RStudio, select **Session**→**Set Working Directory**→**Choose Directory**....

Until the working directory is changed, any files created and saved will automatically be saved in the `Chap1` folder. Once the working directory has been changed, R's workspace is also changed. The workspace in R is known as the global environment, is stored in R as `.GlobalEnv`, and is where any user-defined objects (vectors, matrices, arrays, data frames, lists, functions, etc.) are stored. At the end of an R session, one can save an image of the current workspace that will be automatically reloaded the next time R starts. To list the contents of the workspace, type `ls()` at the R prompt. The contents of the current workspace are also shown in the **Environment** component of RStudio (see Figure 1.8 on the next page). The importance of setting a working directory in R cannot be overstated. This is because all paths in R not beginning with a drive letter on Windows or a leading slash (`/`) on Unix-like systems are relative to the working directory. It is generally best to have a separate directory for each project you start. While one may save the workspace, the authors prefer to keep a record of the commands entered in a script (hence the emphasis on an external editor) so that the workspace can be recreated at a later date.

RStudio will automatically create a folder and set the working directory to the chosen folder when one uses the Project feature. To create a new project, select **File** →**New Project**. Enter a name for the folder where the new project will be created in the Directory name: box, then click the Create Project button (see Figure 1.9 on the following page). The current project name is listed on the far right of the toolbar in a drop-down menu that allows one to switch between projects or create new projects (see Figure 1.8 on the next page where the project name PASWR2E is visible in the far right of the application toolbar).

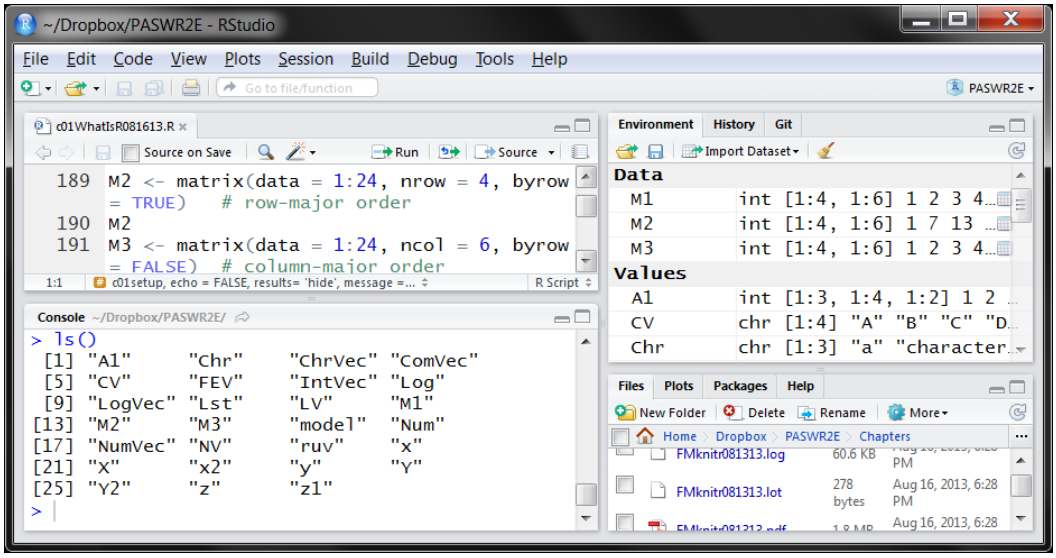

FIGURE 1.8: Contents of the current environment are shown in the top right window in the **Environment** component and in the **Console** component (lower left)

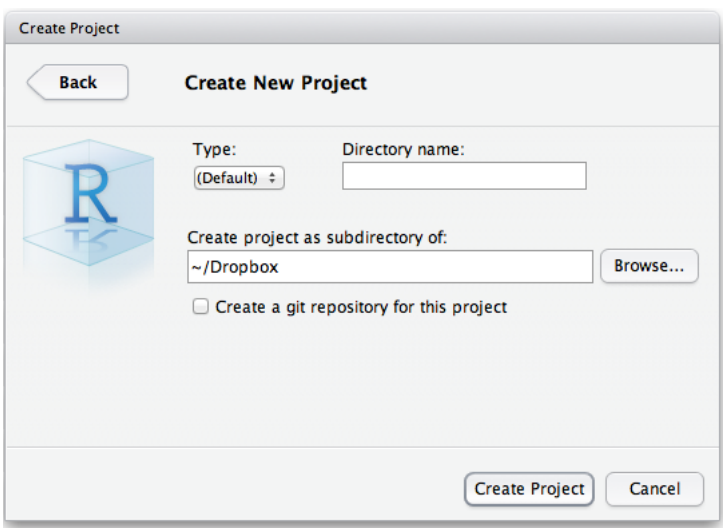

FIGURE 1.9: Create Project dialog for creating a new project

1.8 Packages

Packages are collections of R functions, data, and compiled code that have a uniform organization. All of the data sets, functions, and often the code referenced in this text are available in the package PASWR2. To work through examples and problems, the package PASWR2 should be installed once R is installed. Packages can be installed using the menu interface system on Windows and Mac platforms. On a Windows computer, click **Packages** (the first time you select Packages, you will be prompted to select a CRAN mirror—see Figure 1.10). Then, click **Install Packages(s)** and select the desired package from the menu selection as shown in Figure 1.11 on the next page. Once a package is selected, click **OK**.

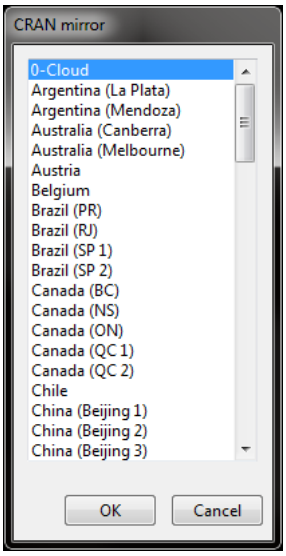

FIGURE 1.10: List from which one selects a **CRAN mirror**

From the *Packages* window, use the scroll bar to find the desired package(s). In this case, select PASWR2; then, click **OK**. On a Mac, click on **Packages & Data**→ **Package Installer**. Select the **Get List** button; then, find the package you would like to install from the list. Before clicking **OK**, select the **Install Dependencies** box so that any packages that are required to run the selected package will also be installed if they are not currently installed. The default setting for Windows machines automatically installs required packages so you do not generally have to worry about this step if you are using the menu interface. If you need to install a package (say PASWR2) at the R command line with a Unix-like operating system or just prefer typing, type `install.packages("PASWR2", dependencies = TRUE)`.

If one is using the RStudio editor, regardless of operating system, one clicks on the **Packages** component, which appears in the lower-right panel of the RStudio editor. The **Packages** component is shown in Figure 1.12 on the following page. Once the component is raised, click the **Install Packages** toolbar button in the lower-right panel. Next, select the checkbox for the packages you would like to install. A package will only need to

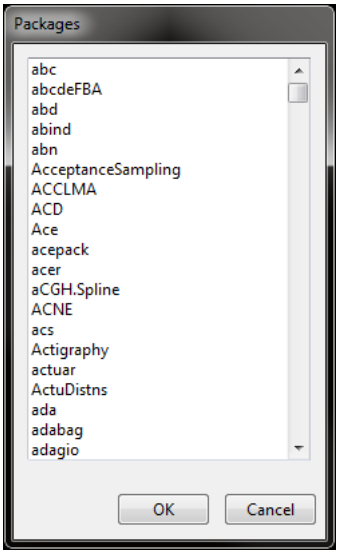

FIGURE 1.11: List of available **Packages**

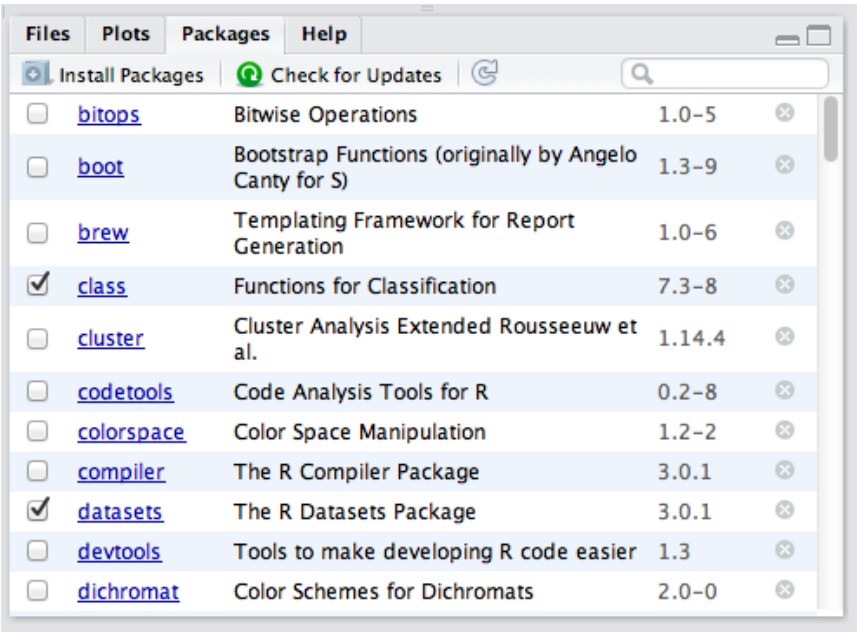

FIGURE 1.12: **Packages** component of RStudio (bottom right)

be installed once; however, to access the contents of a package, R must be told to search a particular path to find the contents of the package. This is done with the command `library("PackageName")`, where `"PackageName"` is the case-sensitive name of the package. Issuing the command `library("PackageName")` is generally referred to as "loading a package." A package can only be loaded if it is installed, and the help files for a package are only available once a package has been loaded. A directory where packages are stored on a computer is called a library. The function `.libPaths()` shows where all the libraries are

located, and the function `library()`, when used without any arguments, lists all available packages in the libraries. To summarize, a package must first be installed, and then the contents of the package are made available by issuing the command `library("PackageName")`.

1.9 R Data Structures

R data structures used in this text include vectors, arrays, matrices, factors, lists, and data frames. Construction of vectors using the `c()`, `seq()`, and `rep()` functions as well as the `:` operator was illustrated earlier. The concept of a vector is crucial, as other data structures are defined in terms of vectors. For example, an array is a vector with a dimension attribute, where the dimension attribute is a vector of non-negative integers; a matrix is also a vector with a dimension attribute (of length two that provides the number of rows and columns); a factor is an encoding of a vector into categories; most R lists are generic vectors; and data frames are data structures similar to matrices that allow the columns (vectors) to be of differing types (numeric, logical, character, etc.). Data frames are the fundamental data structure used for most of R's modeling software and are the primary structure used to archive data in the PASWR2 package.

1.9.1 Arrays and Matrices

Arrays are multidimensional arrangements of elements. A very common example of an array is a matrix, which is a two-dimensional array. The examples in R Code 1.14 show how arrays are constructed from vectors by specifying the arrays' dimensions. The reader should note that the elements of a vector are stored in an array according to the `dim=` attribute argument, which provides the maximal indices in each dimension of the array in row, column, and layer order. For array `A1` (shown in the following code), the values 1 through 24, corresponding to the vector passed to the `data=` argument, are entered down the columns from left to right. This is known as column-major order.

R Code 1.14

```
> A1 <- array(data = 1:24, dim = c(3, 4, 2))
> A1

, , 1

     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12

, , 2

     [,1] [,2] [,3] [,4]
[1,]   13   16   19   22
[2,]   14   17   20   23
[3,]   15   18   21   24

> is.array(A1)
```

```
[1] TRUE

> is.matrix(A1)

[1] FALSE

> class(A1)

[1] "array"

> dim(A1)

[1] 3 4 2
```

Next, a 4-by-6 matrix is constructed and stored in the object `M1`.

```
> M1 <- array(data = 1:24, dim = c(4, 6))
> M1

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24

> is.array(M1)

[1] TRUE

> is.matrix(M1)

[1] TRUE

> class(M1)

[1] "matrix"

> dim(M1)

[1] 4 6
```

Note that `A1` is an array and not a matrix since it has more than two dimensions; however, `M1` is both an array and a matrix, and it has a class designation `"matrix."` Although two-dimensional arrays are matrices and the `array()` function can be used to create a matrix, the `matrix()` function allows the user to create matrices by using row-major order as well as column-major order. When using the function `matrix()`, one may specify the number of rows using the argument `nrow=`; and the number of columns will automatically be computed based on the length of the vector provided to the `data=` argument. Likewise, if one specifies the number of columns using the `ncol=` argument, the number of rows will automatically be computed based on the length of the vector provided to the `data=` argument.

```
> M2 <- matrix(data = 1:24, nrow = 4, byrow = TRUE)  # row-major order
> M2
```

```
     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    2    3    4    5    6
[2,]    7    8    9   10   11   12
[3,]   13   14   15   16   17   18
[4,]   19   20   21   22   23   24

> M3 <- matrix(data = 1:24, ncol = 6, byrow = FALSE)  # column-major order
> M3

     [,1] [,2] [,3] [,4] [,5] [,6]
[1,]    1    5    9   13   17   21
[2,]    2    6   10   14   18   22
[3,]    3    7   11   15   19   23
[4,]    4    8   12   16   20   24
```

In the following example, different types of barley are in the columns; and different provinces in Spain are in the rows. The entries in the matrix represent the weight in thousands of metric tons for each type of barley produced in a given province. The `barley.data` matrix will be used to illustrate various functions and manipulations that can be applied to a matrix. Given the matrix

$$\begin{pmatrix} 190 & 8 & 22.0 \\ 191 & 4 & 1.7 \\ 223 & 80 & 2.0 \end{pmatrix},$$

the values are written to a matrix (reading across the rows with the command `byrow = TRUE`) with the name `barley.data` as follows:

```
> Data <- c(190, 8, 22, 191, 4, 1.7, 223, 80, 2)
> barley.data <- matrix(data = Data, nrow = 3, byrow = TRUE)
> barley.data

     [,1] [,2] [,3]
[1,]  190    8 22.0
[2,]  191    4  1.7
[3,]  223   80  2.0
```

The matrix's dimensions are returned by typing `dim(barley.data)`:

```
> dim(barley.data)

[1] 3 3
```

The function `dim()` applied to `barley.data` in the previous code returns a vector of length two where the first returned value is the number of rows and the second returned value is the number of columns. Consider the following code that creates two objects where the names of the three provinces are assigned to `province` and the three types of barley to `type`:

```
> province <- c("Navarra", "Zaragoza", "Madrid")
> type <- c("typeA", "typeB", "typeC")
```

Assign the names stored in `province` to the rows of the matrix as follows:

```
> dimnames(barley.data) <- list(province, NULL)
> barley.data

         [,1] [,2] [,3]
Navarra   190    8 22.0
Zaragoza  191    4  1.7
Madrid    223   80  2.0
```

Next, assign the names stored in `type` to the columns of the matrix:

```
> dimnames(barley.data) <- list(NULL, type)
> barley.data

     typeA typeB typeC
[1,]   190     8  22.0
[2,]   191     4   1.7
[3,]   223    80   2.0
```

To assign row and column names simultaneously, the command that should be used is `dimnames(barley.data) <- list(province, type)`:

```
> dimnames(barley.data) <- list(province, type)
> barley.data

         typeA typeB typeC
Navarra    190     8  22.0
Zaragoza   191     4   1.7
Madrid     223    80   2.0
```

One can verify the assigned names with the function `dimnames()`:

```
> dimnames(barley.data)

[[1]]
[1] "Navarra"  "Zaragoza" "Madrid"

[[2]]
[1] "typeA" "typeB" "typeC"
```

To delete the row and column name assignments, type

```
> dimnames(barley.data) <- NULL
> barley.data

     [,1] [,2] [,3]
[1,]  190    8 22.0
[2,]  191    4  1.7
[3,]  223   80  2.0
```

If one is interested in only the second row of data, one can enter

```
> dimnames(barley.data) <- list(province, type)
> barley.data[2, ]

typeA typeB typeC
191.0   4.0   1.7
```

or

```
> barley.data["Zaragoza", ]

typeA typeB typeC
191.0   4.0   1.7
```

Note that `barley.data[2, ]` and `barley.data["Zaragoza", ]` are equivalent to typing `barley.data[2, 1:3]` and `barley.data["Zaragoza", 1:3]`. That is, when an index position is left empty, the full range of the index is brought into play. To see the third column, key in

```
> barley.data[, "typeC"]

 Navarra Zaragoza   Madrid
    22.0      1.7      2.0
```

To add an additional column for a fourth type of barley (`typeD`), use the `cbind()` command. Once `typeD` is part of the the `barley.data` data frame, it is removed from the workspace to avoid confusion.

```
> typeD <- c(2, 3.5, 2.75)
> barley.data <- cbind(barley.data, typeD)
> rm("typeD")  # remove typeD from workspace
> barley.data

         typeA typeB typeC typeD
Navarra    190     8  22.0  2.00
Zaragoza   191     4   1.7  3.50
Madrid     223    80   2.0  2.75
```

The function `apply()` allows the user to apply a function to one or more of the dimensions of an array. To calculate the mean of the columns for the matrix `barley.data`, type `apply(X = barley.data, MARGIN = 2, FUN = mean)`, where the value passed to `X` is an array (recall that a matrix is a two-dimensional array). The value of `2` passed to `MARGIN` tells the function to work on the columns. Additionally, the value `mean` passed to `FUN` specifies the function to apply to the respective margin of the array. To read more about `apply()`, type either `?apply` or `help(apply)` at the R prompt.

```
> apply(X = barley.data, MARGIN = 2, FUN = mean)

     typeA      typeB      typeC      typeD
201.333333  30.666667   8.566667   2.750000
```

The second argument, `MARGIN = 2` in the previous example, tells the function `apply()` to work on the columns. For the function to work on rows, the second argument should be `MARGIN = 1`. For example, to find the average barley weight for each province, type

```
> apply(X = barley.data, MARGIN = 1, FUN = mean)

 Navarra Zaragoza   Madrid
 55.5000  50.0500  76.9375
```

The function `names()` allows the assignment of names to vectors:

```
> x <- c(1, 2, 3)
> names(x) <- c("A", "B", "C")
> x

A B C
1 2 3
```

To suppress the names of a vector, type `names(x) <- NULL`:

```
> names(x) <- NULL
> x

[1] 1 2 3
```

Earlier, the text used `barley.data[2, ]` to extract all columns (barley types) for the second row (Zaragoza). The object returned is a vector, not a matrix. This dimensional reduction may seem innocuous at first; however, it may cause problems in code that involves matrix operations. To prevent dimension reduction, one should use the `drop = FALSE` argument. Consider how R Code 1.15 illustrates both the potential problem of dimension reduction as well as code to prevent the dimension reduction.

R Code 1.15

```
> barley.data[2, ]

typeA typeB typeC typeD
191.0   4.0   1.7   3.5

> dim(barley.data[2, ])  # Not a matrix...a vector now

NULL

> is.vector(barley.data[2, ])

[1] TRUE

> barley.data[2, , drop = FALSE]  # A 1*4 matrix not a vector

         typeA typeB typeC typeD
Zaragoza   191     4   1.7   3.5

> is.matrix(barley.data[2, , drop = FALSE])

[1] TRUE

> dim(barley.data[2, , drop = FALSE])

[1] 1 4
```

If one has a vector or a matrix that has been reduced to a vector by mistake and needs to return the object to a matrix, consider using the `as.matrix()` function. When using the `as.matrix()` function, be sure to pay particular attention to the dimension of the resulting matrix. To transpose a matrix, use the `t()` function.

```
> as.matrix(barley.data[2, , drop = FALSE])  # 1*4 matrix

         typeA typeB typeC typeD
Zaragoza   191     4   1.7   3.5

> t(as.matrix(barley.data[2, , drop = FALSE]))  # 4*1 matrix

      Zaragoza
typeA    191.0
typeB      4.0
typeC      1.7
typeD      3.5
```

1.9.2 Vector and Matrix Operations

Consider the system of equations:

$$\begin{aligned} 3\,x + 2\,y + 1\,z &= 10 \\ 2\,x - 3\,y + 1\,z &= -1 \\ 1\,x + 1\,y + 1\,z &= 6. \end{aligned}$$

This system can be represented with matrices and vectors as

$$\mathbf{Ax} = \mathbf{b}, \text{ where } \mathbf{A} = \begin{bmatrix} 3 & 2 & 1 \\ 2 & -3 & 1 \\ 1 & 1 & 1 \end{bmatrix}, \mathbf{x} = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, \text{ and } \mathbf{b} = \begin{bmatrix} 10 \\ -1 \\ 6 \end{bmatrix}.$$

To solve this system of equations, enter **A** and **b** into R and type `solve(A, b)` at the command prompt:

```
> A <- matrix(c(3, 2, 1, 2, -3, 1, 1, 1, 1), byrow = TRUE, nrow = 3)
> A

     [,1] [,2] [,3]
[1,]    3    2    1
[2,]    2   -3    1
[3,]    1    1    1

> b <- matrix(c(10, -1, 6), byrow = TRUE, nrow = 3)
> b

     [,1]
[1,]   10
[2,]   -1
[3,]    6

> x <- solve(A, b)
> x

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

The operator `%*%` is used for matrix multiplication. If $\mathbf{x}$ is an $(n \times 1)$ column vector, and $\mathbf{A}$ is an $(m \times n)$ matrix, then the product of $\mathbf{A}$ and $\mathbf{x}$ is computed by typing `A %*% x`. To verify R's solution, multiply $\mathbf{A} \times \mathbf{x}$, and note that this is equal to $\mathbf{b}$:

```
> A %*% x

     [,1]
[1,]   10
[2,]   -1
[3,]    6
```

Other common functions used with vectors and matrices are included in Table A.2 on page 904.

1.9.3 Factors

A factor is a vector with additional information indicating the distinct values or levels of the vector. Many statistical problems use categorical variables to classify subjects or to subdivide the data. Examples include variables such as socioeconomic status, gender, and marriage status. When using R, one should store categorical data in factors. To create a factor from a vector, one can use the `factor()` function.

Consider a variable where a physician rates how challenging it was for him to find a location for an epidural block as one of the following: easy, difficult, or impossible. For a concrete example, suppose the physician rates four patients where the first is impossible, the second is difficult, the third is easy, and the last patient is rated as impossible. To minimize typing, patients rated as easy, difficult, and impossible are assigned numerical values of 0, 1, and 2, respectively. By using the `levels=` and the `labels=` arguments for the function `factor()`, one can create a factor with three labeled levels as in R Code 1.16.

R Code 1.16

```
> v <- c(2, 1, 0, 2)
> fv <- factor(v)
> fv

[1] 2 1 0 2
Levels: 0 1 2

> fv <- factor(v, levels = 0:2, labels = c("Easy", "Difficult",
+     "Impossible"))
> fv

[1] Impossible Difficult  Easy       Impossible
Levels: Easy Difficult Impossible
```

It is also possible to create a factor from a character vector. One should note that if the `levels=` argument for the `factor()` function is not used with character vectors, the levels of the factor will be sorted alphabetically, which may not be appropriate. In R Code 1.17, the levels of the factor `fv2` are ordered alphabetically as Difficult, Easy, and Impossible.

R Code 1.17

```
> v2 <- c("Impossible", "Difficult", "Easy", "Impossible")
```

```
> fv2 <- factor(v2)
> fv2

[1] Impossible Difficult  Easy       Impossible
Levels: Difficult Easy Impossible
```

To establish the correct levels of ease of palpation, one can use the `levels=` argument or the `levels()` function both illustrated in R Code 1.18.

R Code 1.18

```
> fv3 <- factor(v2, levels = c("Easy", "Difficult", "Impossible"))
> fv3

[1] Impossible Difficult  Easy       Impossible
Levels: Easy Difficult Impossible

> levels(fv2) <- c("Easy", "Difficult", "Impossible")
> fv2

[1] Impossible Easy       Difficult  Impossible
Levels: Easy Difficult Impossible
```

1.9.4 Lists

A list is an object whose elements can be of different modes (character, numeric, logical, etc.). Lists are used to unite related data that have different modes. Since many R functions return results using lists, it is important to know how to extract information from a list. Consider two lists, the first one named `stu1` and the second one named `stu2`. In the first list, tags are used to provide the different objects with names, and the names of the different objects are displayed using the `names()` function. Since tags are not mandatory, the list `stu2` is created without tags, so the reader can better see how to index a list. An individual component, say `major`, of the `stu1` list may be accessed using `$` prefixing or with `[[ ]]` by specifying inside the double square brackets the name of the component in quotes or the index number of the component. To reduce the chance of accessing the wrong component, it is generally better to use the name of the component instead of its index. Creation of the first list (`stu1`) is shown in R Code 1.19. When an entire R expression does not fit on a single line, a `+` symbol appears where the expression wraps to subsequent lines to let the user know the expression is not complete. Once the R expression is complete, the R prompt `>` will reappear in the R console. If the `+` appears in the R console and the user cannot discern how to complete the R expression, a quick solution is to press the escape key, which will return the R prompt but also remove the last expression the user was typing.

R Code 1.19

```
> stu1 <- list(first.name = "Bob", last.name = "Smith",
+         major = "Statistics", semester.hours = 18,
+         grades = c("A", "B+", "A-", "C+", "B", "B-"))
> names(stu1$grades) <- c("Analysis", "Experimental Design", "English",
+                         "German", "Regression", "Programming")
> stu1
```

```
$first.name
[1] "Bob"

$last.name
[1] "Smith"

$major
[1] "Statistics"

$semester.hours
[1] 18

$grades
           Analysis Experimental Design             English
                "A"                "B+"                "A-"
             German          Regression         Programming
               "C+"                 "B"                 "B-"

> names(stu1)

[1] "first.name"     "last.name"      "major"          "semester.hours"
[5] "grades"
```

Creation of the second list (`stu2`) is shown in R Code 1.20.

R Code 1.20

```
> stu2 <- list("Bob", "Smith", "Statistics", 18,
+              c("A", "B+", "A-", "C+", "B", "B-"))
> stu2

[[1]]
[1] "Bob"

[[2]]
[1] "Smith"

[[3]]
[1] "Statistics"

[[4]]
[1] 18

[[5]]
[1] "A"  "B+" "A-" "C+" "B"  "B-"

> names(stu2)  # tags not used

NULL
```

Note that `stu1$major`, `stu1[["major"]]`, and `stu1[[3]]` all return the same value (`"Statistics"`) in R Code 1.21 on the facing page.

R Code 1.21

```
> stu1$major

[1] "Statistics"

> stu1[["major"]]

[1] "Statistics"

> stu1[[3]]

[1] "Statistics"
```

R Code 1.21 illustrates how components are extracted from lists using double square brackets (`[[]]`). When the nth component is a vector, the ith element of the nth component can be extracted using single square brackets (`[]`). One can also extract the ith element of a named component using either `Lst[["name"]][i]` or `Lst$name[i]`, where `Lst` is a list. R Code 1.22 illustrates four approaches to extract the 6th element from the `grades` vector of the `stu1` list.

R Code 1.22

```
> stu1$grades["Programming"]

Programming
       "B-"

> stu1[["grades"]]["Programming"]

Programming
       "B-"

> stu1[[5]]["Programming"]

Programming
       "B-"

> stu1[[5]][6]

Programming
       "B-"
```

1.9.5 Data Frames

A data frame is similar to a matrix in the sense that it is a rectangular structure used to store information. It is different in that all elements of a matrix must be of the same mode (numeric, character, etc.), but this restriction does not apply to data frames. That is, data frames have a two-dimensional structure with rows (experimental units) and columns (variables) where all columns have the same number of rows, which have unique names; yet the columns (variables) in a data frame are not required to be of the same mode. Another way to think of a data frame is as a list with the restriction that all its components are equal length vectors. The data frame is the fundamental data structure used with functions

from the lattice and ggplot2 graphics packages and in most of R's modeling functions. It is also the format used to archive all of the data in the PASWR2 package.

1.9.5.1 Creating Data Frames

The function `data.frame()` can be used to create a data frame by using equal length vectors as the first arguments of the `data.frame()` function. By default, any character variables are converted to factors. If this is undesirable, change the default value of `TRUE` to `FALSE` in the `stringsAsFactor=` argument inside the `data.frame()` function. Unique row names are assigned to the rows of the data frame based on what is passed to the `row.names=` argument. The default argument of `NULL` sequentially numbers the rows starting with a 1.

In R Code 1.23, data frame `DF1` is created. `DF1` uses the default row names. The `str()` function can be used to see the structure of an R object and is used to view the structure of `DF1`. Note that the variable `nv` has class numeric, `cv` has class Factor, and `lv` has class logical. Next, a data frame, `DF2`, is created that uses a character vector supplied to the `row.names=` argument to create unique row names. The `str()` function is also used to view the structure of `DF2`.

R Code 1.23

```
> nv <- c(1, 3, 6, 8)                    # Numeric vector
> cv <- c("a", "d", "f", "p")            # Character vector
> lv <- c(TRUE, FALSE, FALSE, TRUE)      # Logical vector
> DF1 <- data.frame(nv, cv, lv)
> DF1

  nv cv    lv
1  1  a  TRUE
2  3  d FALSE
3  6  f FALSE
4  8  p  TRUE

> str(DF1)

'data.frame': 4 obs. of  3 variables:
 $ nv: num  1 3 6 8
 $ cv: Factor w/ 4 levels "a","d","f","p": 1 2 3 4
 $ lv: logi  TRUE FALSE FALSE TRUE

> DF2 <- data.frame(nv, cv, lv, row.names = c("Joe", "Bob", "Jill", "Sam"),
+                   stringsAsFactors = FALSE)
> DF2

     nv cv    lv
Joe   1  a  TRUE
Bob   3  d FALSE
Jill  6  f FALSE
Sam   8  p  TRUE

> str(DF2)

'data.frame': 4 obs. of  3 variables:
 $ nv: num  1 3 6 8
 $ cv: chr  "a" "d" "f" "p"
 $ lv: logi  TRUE FALSE FALSE TRUE

> rm("nv", "cv", "lv")  # remove the variables from the current environment
```

From the output of R Code 1.23, note that the variable `cv` has class character (`chr`) in `DF2` but has class `Factor` in `DF1` because `stringsAsFactors` was set to `FALSE` in creating `DF2`. One final difference between `DF1` and `DF2` is that `DF2` has names for the rows, that are created by providing a vector of names to the `row.names=` argument in the `data.frame()` function. Row names can be added after the creation of a data frame with the function `row.names()`.

```
> row.names(DF1) <- c("Joe", "Bob", "Jill", "Sam")
> DF1

     nv cv    lv
Joe   1  a  TRUE
Bob   3  d FALSE
Jill  6  f FALSE
Sam   8  p  TRUE
```

1.9.5.2 Accessing Data Frames

Since a data frame is technically a list, its components can be accessed with the same techniques used to access the components of a list, namely `$` prefixing or with `[[ ]]` by specifying inside the double square brackets the name of the component in quotes or the index number of the component. Using the previously created data frame `DF2` and recalling that the variables `nv`, `cv`, and `lv` were removed from the current environment using the `rm()` function, different ways of accessing the information in the variable `nv` are illustrated. Since the individual vectors `nv`, `cv`, and `lv` are no longer present in the current environment, when one enters `nv` at the R prompt, R indicates it cannot find any such object. Seven different ways of accessing the information in the variable `nv` are illustrated in R Code 1.24: dollar (`$`) prefixing , component indexing, component naming, array indexing with names and indices, using the `with()` function, and using the `attach()` function. Note that `DF2[ , "nv"]` is equivalent to `DF2[1:4, "nv"]`. That is, when an index position is left empty, the full range of the index is brought into play.

R Code 1.24

```
> DF2

     nv cv    lv
Joe   1  a  TRUE
Bob   3  d FALSE
Jill  6  f FALSE
Sam   8  p  TRUE

> nv            # nv not on search path it is part of DF2

Error in eval(expr, envir, enclos):  object 'nv' not found

> DF2$nv        # dollar prefixing

[1] 1 3 6 8

> DF2[[1]]      # component indexing

[1] 1 3 6 8
```

```
> DF2[["nv"]]    # component naming

[1] 1 3 6 8

> DF2[, "nv"]    # all rows, column nv

[1] 1 3 6 8

> DF2[, 1]       # all rows, column 1

[1] 1 3 6 8

> with(data = DF2, expr = nv)

[1] 1 3 6 8

> attach(DF2)    # DF2 on search path
> nv

[1] 1 3 6 8

> detach(DF2)    # DF2 removed from search path
> nv             # nv no longer on search path

Error in eval(expr, envir, enclos):  object 'nv' not found
```

If what the user wants is the information in `nv`, simply typing the variable name at the R prompt after attaching the data frame involves less typing than the other four solutions; however, one must exercise care when using the `attach()` function with data frames, especially if any changes are made to the data frame. To understand how R searches for objects and to illustrate the potential danger of using `attach()` after changing the values in a data frame, consider the function `search()`, which returns a list of attached packages and objects. When a data frame is attached using the `attach()` function, it moves to the second position in the search path.

```
> search()      # show attached packages

 [1] ".GlobalEnv"             "package:leaps"
 [3] "package:scatterplot3d" "package:multcomp"
 [5] "package:TH.data"        "package:mvtnorm"
 [7] "package:car"            "package:boot"
 [9] "package:gridExtra"      "package:nortest"
[11] "package:coin"           "package:survival"
[13] "package:binom"          "package:cubature"
[15] "package:plyr"           "package:extrafont"
[17] "package:fontcm"         "package:mapproj"
[19] "package:maps"           "package:xtable"
[21] "package:vcd"            "package:grid"
[23] "package:tikzDevice"     "package:xlsx"
[25] "package:xlsxjars"       "package:rJava"
[27] "package:repmis"         "package:MASS"
[29] "package:PASWR2"         "package:lattice"
[31] "package:ggplot2"        "package:knitr"
```

```
[33] "package:stats"          "package:graphics"
[35] "package:grDevices"      "package:utils"
[37] "package:datasets"       "package:methods"
[39] "Autoloads"              "package:base"

> ls(1)         # shows objects in .GlobalEnv

 [1] "A"            "A1"           "b"            "barley.data" "Chr"
 [6] "ChrVec"       "ComVec"       "CV"           "Data"        "DF1"
[11] "DF2"          "FEV"          "fv"           "fv2"         "fv3"
[16] "IntVec"       "Log"          "LogVec"       "Lst"         "LV"
[21] "M1"           "M2"           "M3"           "model"       "Num"
[26] "NumVec"       "NV"           "province"     "ruv"         "stu1"
[31] "stu2"         "type"         "v"            "v2"          "x"
[36] "X"            "x2"           "y"            "Y"           "Y2"
[41] "z"            "z1"

> attach(DF2)  # place DF2 in search path pos. 2
> search()     # shows DF2 in pos. 2

 [1] ".GlobalEnv"             "DF2"
 [3] "package:leaps"          "package:scatterplot3d"
 [5] "package:multcomp"       "package:TH.data"
 [7] "package:mvtnorm"        "package:car"
 [9] "package:boot"           "package:gridExtra"
[11] "package:nortest"        "package:coin"
[13] "package:survival"       "package:binom"
[15] "package:cubature"       "package:plyr"
[17] "package:extrafont"      "package:fontcm"
[19] "package:mapproj"        "package:maps"
[21] "package:xtable"         "package:vcd"
[23] "package:grid"           "package:tikzDevice"
[25] "package:xlsx"           "package:xlsxjars"
[27] "package:rJava"          "package:repmis"
[29] "package:MASS"           "package:PASWR2"
[31] "package:lattice"        "package:ggplot2"
[33] "package:knitr"          "package:stats"
[35] "package:graphics"       "package:grDevices"
[37] "package:utils"          "package:datasets"
[39] "package:methods"        "Autoloads"
[41] "package:base"

> ls(2)         # shows objects in pos. 2 (DF2)

[1] "cv" "lv" "nv"
```

Suppose the values in `nv` have been systematically recorded incorrectly such that the correct values should be the current values plus 5. One may be tempted to change the values with the code `nv <- nv + 5`. Doing this is not a good solution because it creates another `nv` object in R's search path that is not equivalent to the `nv` object in position 2. Observe how the `nv` stored in the global environment (position 1 of the search path) and the object `nv` of the `DF2` data frame (position 2 of the search path) have different values.

```
> nv <- nv + 5  # nv stored in workspace
> nv            # nv from pos. 1

[1]  6  8 11 13

> ls(2)         # list objects in position 2

[1] "cv" "lv" "nv"

> DF2$nv        # nv from pos. 2

[1] 1 3 6 8

> detach(DF2)   # remove DF2 from search path
> search()      # show attached packages

 [1] ".GlobalEnv"            "package:leaps"
 [3] "package:scatterplot3d" "package:multcomp"
 [5] "package:TH.data"       "package:mvtnorm"
 [7] "package:car"           "package:boot"
 [9] "package:gridExtra"     "package:nortest"
[11] "package:coin"          "package:survival"
[13] "package:binom"         "package:cubature"
[15] "package:plyr"          "package:extrafont"
[17] "package:fontcm"        "package:mapproj"
[19] "package:maps"          "package:xtable"
[21] "package:vcd"           "package:grid"
[23] "package:tikzDevice"    "package:xlsx"
[25] "package:xlsxjars"      "package:rJava"
[27] "package:repmis"        "package:MASS"
[29] "package:PASWR2"        "package:lattice"
[31] "package:ggplot2"       "package:knitr"
[33] "package:stats"         "package:graphics"
[35] "package:grDevices"     "package:utils"
[37] "package:datasets"      "package:methods"
[39] "Autoloads"             "package:base"
```

To change the values of the variable `nv` for the data frame `DF2`, one might use something like `DF2$nv <- DF2$nv + 5`.

```
> DF2$nv <- DF2$nv + 5  # nv changed inside DF2
> DF2

     nv cv    lv
Joe   6  a  TRUE
Bob   8  d FALSE
Jill 11  f FALSE
Sam  13  p  TRUE
```

While the `attach()` function will often be used for convenience, be aware that changes to variables of attached data frames are not saved in the attached data frame. It is a good practice to remove a data frame from the search path using the `detach()` function when you no longer need the data frame. It is also possible to return the ith column(s) of a data

frame as a data frame using single brackets (`[]`) by providing a vector specifying the desired columns. If array indexing is used, the default argument `drop = TRUE` must be changed to `drop = FALSE` as illustrated in R Code 1.25.

R Code 1.25

```
> DF2[c(1, 3)]  # extract 1st and 3rd columns

     nv    lv
Joe   6  TRUE
Bob   8 FALSE
Jill 11 FALSE
Sam  13  TRUE

> DF2[c("nv", "lv")]  # extract columns named nv and lv

     nv    lv
Joe   6  TRUE
Bob   8 FALSE
Jill 11 FALSE
Sam  13  TRUE

> DF2[, c(1, 3), drop = FALSE]  # extract all rows for 1st and 3rd columns

     nv    lv
Joe   6  TRUE
Bob   8 FALSE
Jill 11 FALSE
Sam  13  TRUE

> DF2[, c("nv", "lv"), drop = FALSE]  # all rows for columns nv and lv

     nv    lv
Joe   6  TRUE
Bob   8 FALSE
Jill 11 FALSE
Sam  13  TRUE
```

It is also possible to use the convenience function `subset()` as follows.

```
> subset(x = DF2, select = c(nv, lv))

     nv    lv
Joe   6  TRUE
Bob   8 FALSE
Jill 11 FALSE
Sam  13  TRUE
```

1.9.5.3 Accessing Data from Packages

Package authors may store their data in many different formats, with data frames being one of the more common formats. To access data sets from a package, the package must first be in the search path. To place a package, say PackageName, in the search

path, one should use the `library()` function (`library(PackageName)`). Removal of a package, say PackageName, from the search path is done with the `detach()` function (`detach(package:PackageName)`). Once a package is in the search path, its contents can generally be viewed by typing `data()` at the R prompt. To view a short description of all available data sets installed on your machine, type `data()` at the R prompt (see Figure 1.13). The data sets of a particular package, provided it is installed, can be viewed by entering `data(package="PackageName")` where PackageName is the case-sensitive name of the desired package. To view the data sets for all installed packages, enter `data(package = .packages(all.available = TRUE))`.

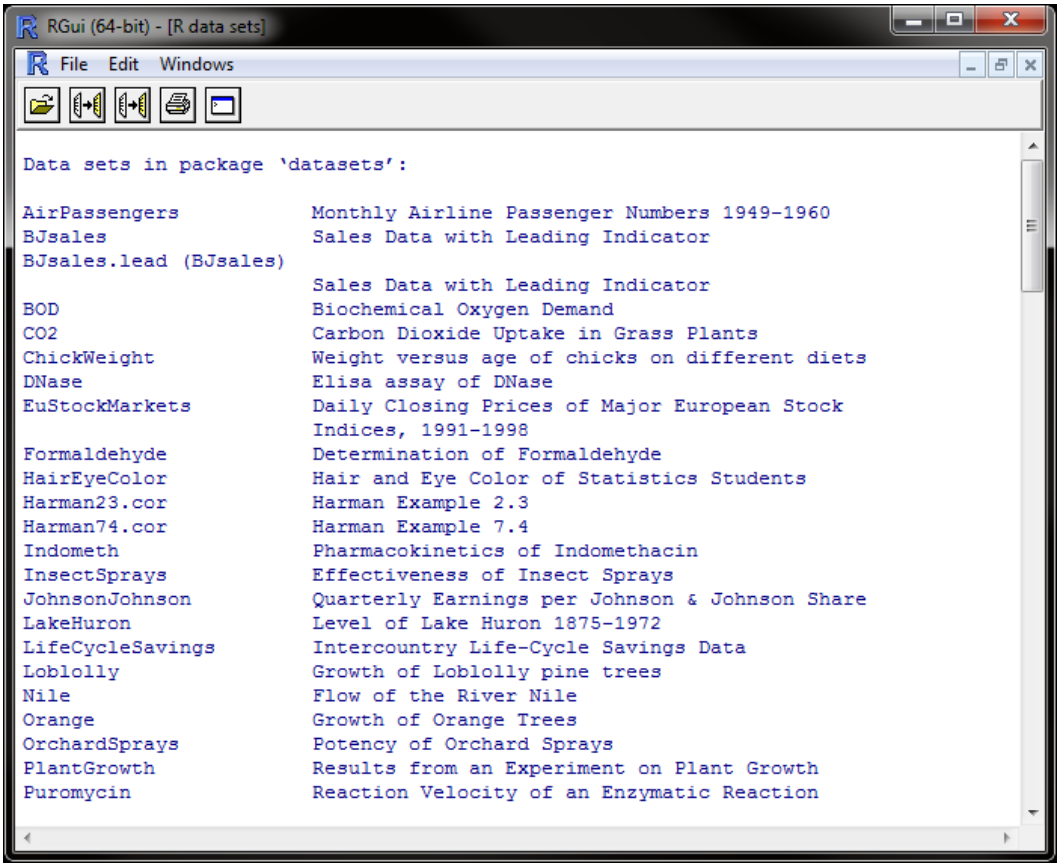

FIGURE 1.13: Available data sets

R Code 1.26 places the package MASS in the search path, opens an HTML help file (as in Figure 1.14 on the facing page) where information is provided about the data frame `Animals`, and uses the function `head(Animals)` to show the first six rows of the data frame. To view the last six rows of a data frame, use the `tail()` function. To view a different number of rows other than the default six with the functions `head()` and `tail()`, use the argument `n=` to specify the number of rows one desires to view.

R Code 1.26

```
> library(MASS)  # Places MASS in search path
```

```
> help(Animals)  # Opens HTML help window
> head(Animals)  # shows first six rows

                    body brain
Mountain beaver     1.35   8.1
Cow               465.00 423.0
Grey wolf          36.33 119.5
Goat               27.66 115.0
Guinea pig          1.04   5.5
Dipliodocus     11700.00  50.0
```

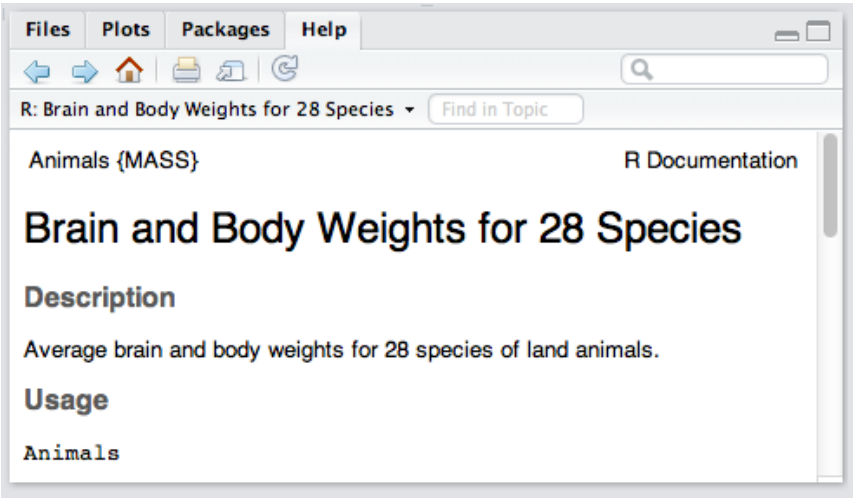

FIGURE 1.14: **Help** component showing the HTML help file for the `Animals` data frame from the MASS package

While most of the data in this text is stored in the PASWR2 package, the reader may need to work with data stored in external files in a variety of different formats. For example, the reader may have data stored in a spreadsheet as an `.xls`, `.xslx`, or `.csv` file, or one may want to read in data stored as text in a `.txt` file. R can read many different types of files and the formats that are most useful for the material in this text will be covered.

1.10 Reading and Saving Data in R

R has the ability to read data from external files stored in several formats. To read data in formats other than ASCII, the package foreign can be used. This package is able to read formats from other statistical programs such as EpiInfo, Minitab, S-Plus, SAS, SPSS, Stata, and Systat. For more information on reading data from other statistical programs, relational databases, and binary files, please reference the *R Data Import/Export Manual* available online and accessible from the **R HTML** help menu or the **Help** component of

RStudio. For all but the smallest of data sets, when working with data stored in a format not readable by R, it will almost always prove easier first to save the original data as a text file and then to read the external file using `read.table()` or `scan()`. Typically, `read.table()` is more user friendly, although `scan()` reads large data of a single mode more quickly than does `read.table()`. It also reads data from the console. To download a wide variety of files from the Internet, one can use the function `download.file()`, which allows the user to specify where they want the downloaded file saved by using the `destfile=` argument.

1.10.1 Using `read.table()`

The function `read.table()` reads a file in table format (a rectangular data set) and creates a data frame from that file. The file may be on your computer, at an unsecure (`http`) website or, for Windows users, the contents of the clipboard (`file = "clipboard"`). To create an R data frame from an external file, one can use the function `read.table()` or one of its variants. Consider the text file `FAT.txt` available online from the website: http://www1.appstate.edu/~arnholta/PASWR/CD/data/Bodyfat.txt.

```
> site <- "http://www1.appstate.edu/~arnholta/PASWR/CD/data/Bodyfat.txt"
> FAT <- read.table(file = site, header = TRUE, sep = "\t")
> head(FAT)     # Show first six rows

  age  fat sex
1  23  9.5   M
2  23 27.9   F
3  27  7.8   M
4  27 17.8   M
5  39 31.4   F
6  41 25.9   F
```

The website's address is stored in the R object `site`. This is done so that the `read.table()` command can appear on a single line. Inside the function `read.table()`, the only mandatory argument is the file location. Since `Bodyfat.txt` has column names and is a tab-delimited file, the arguments `header = TRUE` and `sep = "\t"` are used. There are numerous arguments for `read.table()`: the field separator character (`sep=`), which by default is set to a blank space; the character used for decimal points (`dec=`); the character vector used to represent missing values (`na.strings=`); and many others. To read about all of the arguments for `read.table()` and its variants such as `read.csv()`, which can be used to read comma-separated value (`.csv`) files, type `?read.table` at the R prompt. If the `Bodyfat.txt` file were stored in the current working directory of R, then one would only need to specify the file name in quotes of the `file=` argument. When files are not in the working directory, then the complete path to the desired file must be used.

Microsoft Excel spreadsheets are often stored as `.csv` files. For American readers, no confusion should result when viewing a `.csv` file, as the values are separated by commas just as the name implies. On the other hand, in certain countries, Excel will save the contents of a `.csv` spreadsheet using the semicolon (;) as the field separator and the comma (,) as the decimal point. Users in these locales may find the function `read.csv2()` useful, as its default arguments for field separation and decimal points are `sep = ";"` and `dec = ","`, respectively. To open the `Bodyfat.csv` file stored on the Internet, type

```
> site <- "http://www1.appstate.edu/~arnholta/PASWR/CD/data/Bodyfat.csv"
> FAT <- read.csv(file = site)
> head(FAT)  # Show first six rows
```

```
  age  fat sex
1  23  9.5   M
2  23 27.9   F
3  27  7.8   M
4  27 17.8   M
5  39 31.4   F
6  41 25.9   F
```

Since the default argument for the header is `header = TRUE` in `read.csv()`, only the file was specified in the function.

1.10.2 Using `download.file()`

The function `download.file()` downloads files from the Internet and allows the user to specify where they want the downloaded file saved by providing a character string specifying the path, including the name of the downloaded file to be saved to the `destfile` argument. This function is especially useful when the file is large and/or the user's Internet connection is slow. Consider downloading the gross salaries for Baltimore city employees, which are available as one of the 110 publicly available data sets at https://data.baltimorecity.gov, and storing the results in a directory named `Data`. R Code 1.27 uses the `paste0()` function to concatenate two strings, each on a separate line, into a single string. The single string would extend beyond the allowable margins of the text; consequently, `paste0()` is used for cosmetic purposes.

R Code 1.27

```
> site <- paste0("http://data.baltimorecity.gov/api/",
+                "views/7ymi-bvp3/rows.csv?accessType=DOWNLOAD")
> download.file(site, destfile = "./Data/Salaries.csv")
> list.files("./Data")                      # show files in ./Data

[1] "Salaries.csv"

> file.info("./Data/Salaries.csv")          # file information

                       size isdir mode               mtime
./Data/Salaries.csv 1524254 FALSE  644 2015-06-02 09:12:27
                                  ctime               atime uid gid uname
./Data/Salaries.csv 2015-06-02 09:12:27 2015-06-02 08:09:21 501  20  alan
                    grname
./Data/Salaries.csv  staff

> BES <- read.csv("./Data/Salaries.csv")
> head(BES, n = 2)                          # show first two rows

              name                      JobTitle AgencyID
1 Aaron,Patricia G Facilities/Office Services II   A03031
2    Aaron,Petra L    ASSISTANT STATE'S ATTORNEY   A29005
                    Agency   HireDate AnnualSalary  GrossPay
1      OED-Employment Dev  10/24/1979    $51862.00 $52247.39
2 States Attorneys Office  09/25/2006    $64000.00 $59026.81
```

The function `file.info()` shows the size of the downloaded file as 1,524,254 bytes or 1.524254 megabytes and the date and time the file was last accessed under `atime`.

The function `download.file()` is not restricted to reading `.csv` files. Although the authors prefer to use human readable files instead of binary files, large files may need to be compressed and stored as `.zip` files. Consider using `download.file()` to download a zipped version of the Baltimore city employees salary data downloaded December 5, 2014, and stored at http://bit.ly/12H9E0l, which is the shortened version of the original URL: http://www1.appstate.edu/~arnholta/PASWR/CD/data/Salaries.csv.zip.

```
> site <- "http://bit.ly/12H9E0l"  # URL
> download.file(site, destfile = "./DataZ/Salaries.csv.zip")
> file.info("./DataZ/Salaries.csv.zip")['size']

                            size
./DataZ/Salaries.csv.zip 419937
```

Note that the zipped file is 419,937 bytes while the `.csv` file is 1,524,254 bytes. By using a compression algorithm, the zipped file is 3.6297 times the size of the `.csv` file.

1.10.3 Reading Data from Secure Websites

Secure websites start with `https`, in contrast to unsecure websites such as those used in Section 1.10.1, which begin with `http`. File-sharing services such as Dropbox and GitHub store their data on secure websites. One approach to reading a data file from a secure website is to use the function `source_data()` from the R package repmis (Gandrud, 2015). To read a data file into R from the *Public* folder on Dropbox, use `source_data()`; to read a data file into R from a non-*Public* folder on Dropbox, use the function `source_DropboxData()`. Dropbox has a *Public* folder on all accounts created prior to October 4, 2012. If one is using a Dropbox account created after October 4, 2012, see the website https://www.dropbox.com/help/16/en for directions on how to create a *Public* folder.

Consider the file `Verizon.csv` (Chihara and Hesterberg, 2011) stored in the *Public* folder of a Dropbox account, https://db.tt/1rlTfYnk, which is the shortened version of the original URL: https://dl.dropboxusercontent.com/u/134274843/data/Verizon.csv. R Code 1.28 reads the data into the data frame `Verizon1`. The same data file, `Verizon.csv`, is also stored in a non-*Public* folder on Dropbox http://bit.ly/1mYM0jV, which is the shortened version of the original URL: https://www.dropbox.com/s/a9muo5wybukfs86/Verizon.csv.

R Code 1.28

```
> library(repmis)
> URL <- "https://db.tt/1rlTfYnk"
> Verizon1 <- source_data(url = URL, sep = ",", header = TRUE)

Downloading data from:  https://db.tt/1rlTfYnk
   SHA-1 hash of the downloaded data file is:
6a26836a830af142c3562a6e4fe612eeb0281c30

> head(Verizon1)  # show first six rows of data

   Time Group
1 17.50  ILEC
2  2.40  ILEC
3  0.00  ILEC
```

```
4  0.65  ILEC
5 22.23  ILEC
6  1.20  ILEC
```

Inside R Code 1.28, after the third line of typed code, one sees three lines of italicized text. The long string of numbers and letters on the third line of italicized text after *SHA-1 hash of the downloaded data file is:* is a unique identifier assigned by the function `source_data()` to the data file. The identifier allows the user to verify that the data file they downloaded is indeed the data file desired. That is, the unique identifier will change if the contents of the data file change. One can verify that the data downloaded in R Code 1.28 and the data downloaded in R Code 1.29 and R Code 1.30 are indeed identical. There is only one required argument for `source_data()`, `url=`. The user must provide a universal resource locator (URL, that is a website) to the argument `url=` for `source_data()` to work. Data files at unsecure websites as well as secure websites can be read into R with `source_data()`.

To download data from GitHub, one may also use `source_data()`. Caution needs to be exercised to make sure the data file one downloads is a "raw" text file. When one navigates to a directory on GitHub that contains data, the initial view may contain HTML embedded in the data file. To get a plain text file, click on the "raw" button in the upper right portion of the GitHub window. R Code 1.29 reads the file `Verizon.csv` from GitHub http://bit.ly/1gqZCX3, which is the shortened version of the original URL https://raw.githubusercontent.com/alanarnholt/Data/master/Verizon.csv, and stores the result in the data frame `Verizon2`.

R Code 1.29

```
> URL <- "http://bit.ly/1gqZCX3"
> Verizon2 <- source_data(url = URL)

Downloading data from:  http://bit.ly/1gqZCX3
   SHA-1 hash of the downloaded data file is:
6a26836a830af142c3562a6e4fe612eeb0281c30

> head(Verizon2)  # show first six rows of data

   Time Group
1 17.50  ILEC
2  2.40  ILEC
3  0.00  ILEC
4  0.65  ILEC
5 22.23  ILEC
6  1.20  ILEC
```

Since the default arguments for `source_data()` are `sep = ","` and `header = TRUE`, which read in a `.csv` file with a header, neither argument was used as they were in R Code 1.28.

To read the `Verizon.csv` data file, which is also stored in a non-*Public* folder, one can use the function `source_DropboxData()` as illustrated in R Code 1.30. There are two required arguments for `source_DropboxData()`: the data file's name and the data file's Dropbox key. To find a file's Dropbox key, right click on the data file's name; then, click on **Share Dropbox Link** from the drop-down menu. The link to the selected file will be copied to the clipboard. The Dropbox link is a URL that can be pasted into a web browser. The URL referenced in R Code 1.30 is

https://www.dropbox.com/s/a9muo5wybukfs86/Verizon.csv

The last part of the URL, `Verizon.csv`, is the data file's name. The Dropbox key is the string of letters and numbers just after `htpps://www.dropbox.com/s/`, which in this case is `a9muo5wybukfs86`.

R Code 1.30

```
> Verizon3 <- source_DropboxData("Verizon.csv", "a9muo5wybukfs86")
> head(Verizon3)

   Time Group
1 17.50  ILEC
2  2.40  ILEC
3  0.00  ILEC
4  0.65  ILEC
5 22.23  ILEC
6  1.20  ILEC
```

1.10.4 Using `scan()`

The function `scan()` works well for entering a small amount of data by either typing in the console or by using a combination of copying and pasting procedures when the data can be highlighted and copied. To enter the ages for the subjects in the previously created `FAT` data frame whose values are also shown in Table 1.1 on page 58, one can proceed in two ways. One can enter all of the ages in one row, or one can enter one age per row. Regardless of how the data is entered, input is terminated with a blank line or an end of file signal (`Ctrl-Z` on Windows and `Ctrl-D` on a Mac). While it is possible to use `scan()` to read a file, the data sets in this text are generally tabular and `read.table()` and its variants are specifically designed to read in tabular data.

```
> age1 <- scan()
1: 23 23 27 27 39 41 45 49 50 53 53 54 56 57 58 58 60 61
19:
Read 18 items
> age1
 [1] 23 23 27 27 39 41 45 49 50 53 53 54 56 57 58 58 60 61
> age2 <-scan()
1: 23
2: 23
3: 27
.
.
.
18: 61
19:
Read 18 items
> age2
 [1] 23 23 27 27 39 41 45 49 50 53 53 54 56 57 58 58 60 61
```

1.10.5 Reading Excel (.xlsx) Files

The xlsx package, which must be installed, gives programmatic control of Excel files using R. The function `read.xlsx()` can be used to read an Excel workbook stored as a `.xlsx` file. When the workbook consists of multiple worksheets, the argument `sheetName=` is used to specify the desired worksheet. Consider an Excel workbook named `faculty.xlsx` stored in R's current working directory that has two worksheets, `Univ1` and `Univ2`. See Figures 1.15 and 1.16 on page 47 to view the contents of the Excel worksheets as they appear inside Excel. R Code 1.31 reads the worksheet named `Univ1` from the Excel file `faculty.xlsx`, stored in a directory named `Data` one level up from the working directory, into a data frame named `Faculty1`, which is then shown in the console. Next, the worksheet named `Univ2` from the same workbook is read into a data frame named `Faculty2` and subsequently displayed in the console.

R Code 1.31

```
> library(xlsx)  # Loading xlsx package
> Faculty1 <- read.xlsx(file = "../Data/FACULTY.xlsx", sheetName = "Univ1")
> Faculty1

   Name Height      Rank
1   Joe     72 Assistant
2 Susie     63 Professor
3    Al     74 Associate
4   Rob     69 Professor
5  Juan     65 Professor

> Faculty2 <- read.xlsx(file = "../Data/FACULTY.xlsx", sheetName = "Univ2")
> Faculty2

   Name Height      Rank
1  Lola     62 Professor
2   Ana     61 Professor
3 Maria     65 Associate
4 Pilar     69 Assistant
5   Eva     65  Lecturer
```

It is worth reiterating the fact that one should always set R's working directory. Without the proper working directory, the previous code would not create the objects where the user wants them. Since R allows relative paths to be used in the `file=` argument, one has an additional incentive to establish a working directory so that code the user writes is portable across machines. A relative path is one that is defined in relation to the current or working directory. This means that a relative path on Windows will not start with a drive letter; and on Mac and Unix-like operating systems, a relative path will not start with a forward slash (`/`). To refer to `SomeFile` in the current directory, use a single dot (`./SomeFile`). Moving up the file system is done with two dots (`..`). To refer to `SomeFile` two levels above the working directory, type `../../SomeFile`.

The package `repmis` has the function `source_XlsxData()`, which can read Excel files stored at a URL (both http and https) into R. R Code 1.32 reads the file `FACULTY.xlsx` from GitHub http://bit.ly/1iOWsGP, which is the shortened version of the original URL https://github.com/alanarnholt/Data/raw/master/FACULTY.xlsx, and stores the results from sheet `"Univ1"` in the data frame `Faculty3` and sheet `"Univ2"` in the data frame `Faculty4`.

R Code 1.32

```
> URL <- "http://bit.ly/1iOWsGP"
> Faculty3 <- repmis::source_XlsxData(url = URL, sheet = "Univ1")

Downloading data from:  http://bit.ly/1iOWsGP
   SHA-1 hash of the downloaded data file is:
5beb512f1dbcb421fd1bd315bc9b8b2be4bb00ec

> Faculty4 <- repmis::source_XlsxData(url = URL, sheet = "Univ2")

Downloading data from:  http://bit.ly/1iOWsGP
   SHA-1 hash of the downloaded data file is:
5beb512f1dbcb421fd1bd315bc9b8b2be4bb00ec
```

```
> Faculty3

   Name Height      Rank
1   Joe     72 Assistant
2 Susie     63 Professor
3    Al     74 Associate
4   Rob     69 Professor
5  Juan     65 Professor

> Faculty4

   Name Height      Rank
1  Lola     62 Professor
2   Ana     61 Professor
3 Maria     65 Associate
4 Pilar     69 Assistant
5   Eva     65  Lecturer
```

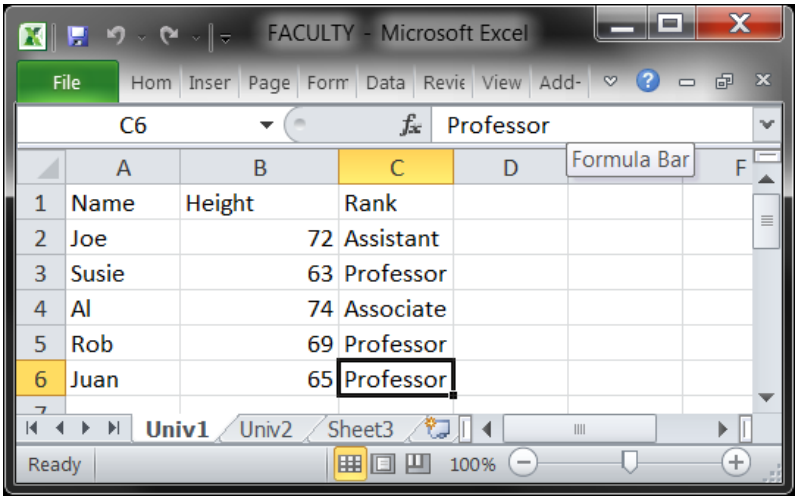

FIGURE 1.15: Excel workbook `faculty.xlsx` worksheet 1 contents

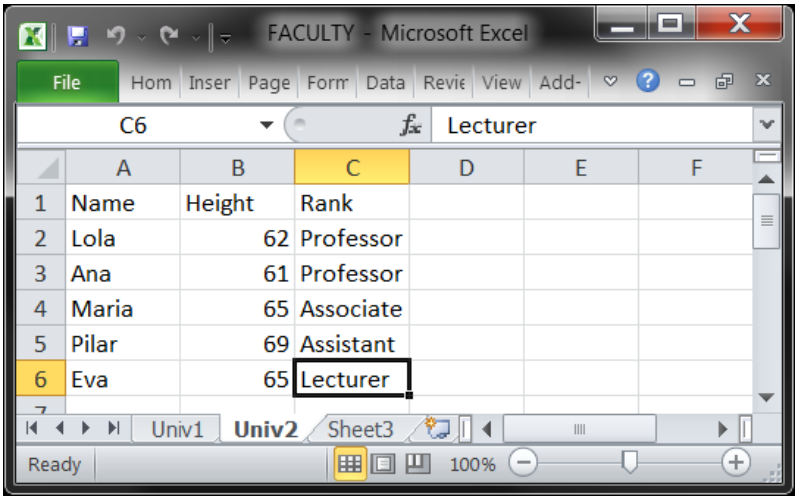

FIGURE 1.16: Excel workbook `faculty.xlsx` worksheet 2 contents

1.10.6 Saving Data Frames to External Files

The function `write.table()` writes an R data frame to a file. Just as `read.table()` had variants `read.csv()` and `read.csv2()`, `write.table()` has variants `write.csv()` and `write.csv2()` to write to a `.csv` file. To write the `FAT` data frame, stored in the global environment, to the file `FAT.txt` in R's current working directory, type

```
> write.table(FAT, file = "FAT.txt")
```

The previous command, by default, stores the data frame `FAT` to the file `FAT.txt` using blank spaces as the separators. To store the file using tab separation, key in

```
> write.table(FAT, file = "FAT.txt", sep = "\t")
```

To see the contents of the file `FAT.txt` in R, one can use the function `file.show()`. The result from using `file.show("FAT.txt")` on a tab delimited file can be seen in Figure 1.17 on the next page. To store a data frame to a file outside of the current working directory, the path either relative or absolute to the desired location must be given to the `file=` argument. The write analog to `read.xlsx()` is `write.xlsx()`. To write the `FAT` data frame stored in the global environment to the file `FAT.xlsx` in R's current working directory, type

```
> write.xlsx(FAT, file = "FAT.xlsx")
```

1.11 Working with Data

This section presents a data set that shows how different data types should be read into R as well as several functions that are useful for working with different types of R objects. Consider the data stored as a `.csv` file at

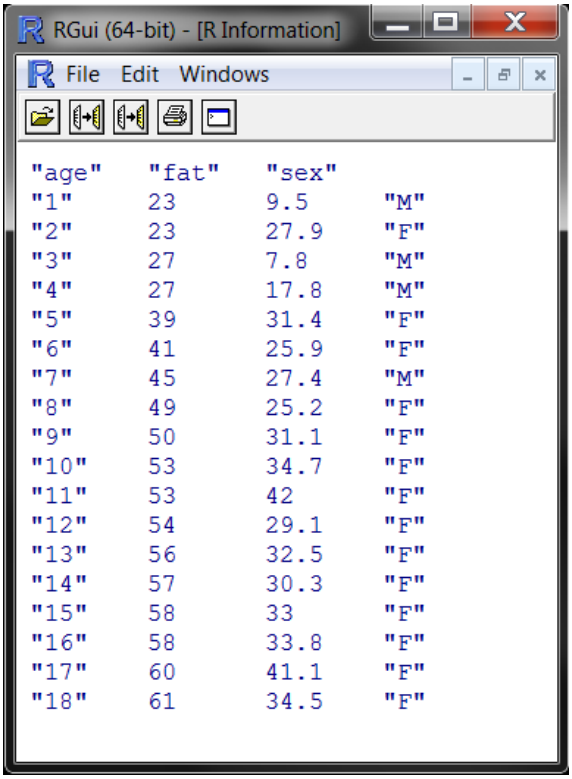

FIGURE 1.17: Results of `file.show("FAT.txt")`

http://www1.appstate.edu/~arnholta/PASWR/CD/data/Poplar3.CSV.

The following description of the data is from Minitab 15:

> In an effort to maximize yield, researchers designed an experiment to determine how two factors, Site and Treatment, influence the Weight of four-year-old poplar clones. They planted trees on two sites: Site 1 is a moist site with rich soil, and Site 2 is a dry, sandy site. They applied four different treatments to the trees: Treatment 1 was the control (no treatment); Treatment 2 used fertilizer; Treatment 3 used irrigation; and Treatment 4 use both fertilizer and irrigation. To account for a variety of weather conditions, the researchers replicated the data by planting half the trees in Year 1, and the other half in Year 2.

The data from `Poplar3.CSV` is read into the data frame `poplar` using the `read.csv()` function, and the first five rows of the data frame are shown using the function `head()` with the argument `n = 5` to show the first five rows of the data frame instead of the default `n = 6` rows in R Code 1.33.

R Code 1.33

```
> site <- "http://www1.appstate.edu/~arnholta/PASWR/CD/data/Poplar3.CSV"
> poplar <- read.csv(file = url(site))
> head(poplar, n = 3)  # show first three rows

  Site Year Treatment Diameter Height Weight Age
```

```
1     1     1          1      2.23    3.76    0.17    3
2     1     1          1      2.12    3.15    0.15    3
3     1     1          1      1.06    1.85    0.02    3
```

When dealing with imported data sets, it is always good to examine their contents using functions such as `str()` and `summary()`, which show the structure and provide appropriate summaries, respectively, for different types of objects.

```
> str(poplar)

'data.frame': 298 obs. of  7 variables:
 $ Site     : int  1 1 1 1 1 1 1 1 1 2 ...
 $ Year     : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Treatment: int  1 1 1 1 1 1 1 1 1 1 ...
 $ Diameter : num  2.23 2.12 1.06 2.12 2.99 4.01 2.41 2.75 2.2 4.09 ...
 $ Height   : num  3.76 3.15 1.85 3.64 4.64 5.25 4.07 4.72 4.17 5.73 ...
 $ Weight   : num  0.17 0.15 0.02 0.16 0.37 0.73 0.22 0.3 0.19 0.78 ...
 $ Age      : int  3 3 3 3 3 3 3 3 3 3 ...

> summary(poplar)

      Site           Year        Treatment        Diameter
 Min.   :1.00   Min.   :1.00   Min.   :1.000   Min.   :-99.000
 1st Qu.:1.00   1st Qu.:1.00   1st Qu.:2.000   1st Qu.:  3.605
 Median :2.00   Median :2.00   Median :2.500   Median :  5.175
 Mean   :1.51   Mean   :1.51   Mean   :2.503   Mean   :  3.862
 3rd Qu.:2.00   3rd Qu.:2.00   3rd Qu.:3.750   3rd Qu.:  6.230
 Max.   :2.00   Max.   :2.00   Max.   :4.000   Max.   :  8.260
     Height            Weight             Age
 Min.   :-99.000   Min.   :-99.000   Min.   :3.000
 1st Qu.:  5.495   1st Qu.:  0.605   1st Qu.:3.000
 Median :  6.910   Median :  1.640   Median :4.000
 Mean   :  5.902   Mean   :  1.099   Mean   :3.507
 3rd Qu.:  8.750   3rd Qu.:  3.435   3rd Qu.:4.000
 Max.   : 10.900   Max.   :  6.930   Max.   :4.000
```

From typing `str(poplar)` at the R prompt, one can see that all seven variables are either integer or numeric. From the description, the variables `Site` and `Treatment` are factors. Further investigation into the experiment reveals that `year` and `Age` are factors as well. Recall that factors are an extension of vectors designed for storing categorical information. The results of `summary(poplar)` indicate the minimum values for `Diameter`, `Height`, and `Weight` are all -99, which does not make sense unless one is told that a value of -99 for these variables represents a missing value. Once one understands that the variables `Site`, `Year`, `Treatment`, and `Age` are factors and that the value -99 has been used to represent missing values for the variables `Diameter`, `Height`, and `Weight`, appropriate arguments to `read.csv()` can be entered. The data is now read into the object `poplarC` using `na.strings = "-99"` to store the `NA` values correctly. The argument `colClasses=` requires a vector that indicates the desired class of each column.

```
> poplarC <- read.csv(file = url(site), na.strings = "-99",
+     colClasses = c(rep("factor", 3), rep("numeric", 3), "factor"))
> str(poplarC)
```

```
'data.frame': 298 obs. of  7 variables:
 $ Site     : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 2 ...
 $ Year     : Factor w/ 2 levels "1","2": 1 1 1 1 1 1 1 1 1 1 ...
 $ Treatment: Factor w/ 4 levels "1","2","3","4": 1 1 1 1 1 1 1 1 1 1 ...
 $ Diameter : num  2.23 2.12 1.06 2.12 2.99 4.01 2.41 2.75 2.2 4.09 ...
 $ Height   : num  3.76 3.15 1.85 3.64 4.64 5.25 4.07 4.72 4.17 5.73 ...
 $ Weight   : num  0.17 0.15 0.02 0.16 0.37 0.73 0.22 0.3 0.19 0.78 ...
 $ Age      : Factor w/ 2 levels "3","4": 1 1 1 1 1 1 1 1 1 1 ...
```

In the event different values (999, 99, 9999) for different variables (`var1`, `var2`, `var3`) are used to represent missing values in a data set, the argument `na.strings=` will no longer be able to solve the problem directly. Although one can pass a vector of the form `na.strings = c(999, 99, 9999)`, this will simply replace all values that are 999, 99, or 9999 with `NA`s. If the first variable has a legitimate value of 99, then it too would be replaced with an `NA` value. One solution for this problem in general is to read the data set into a data frame (`DF`), to assign the data frame to a different name so that the cleaned up data set is not confused with the original data, and to use filtering to assign `NA`s to values of `var1`, `var2`, and `var3` that have entries of 999, 99, and 999, respectively.

```
> DF <- read.table(file = url(site), header = TRUE)
> df <- DF
> df[df$var1 == 999, "var1"] = NA
> df[df$var2 == 99, "var2"] = NA
> df[df$var3 == 9999, "var3"] = NA
```

Once a variable has its class changed from `int` to `factor`, labeling the levels of the factor can be accomplished without difficulties. To facilitate analysis of the `poplarC` data, labels for the levels of the variables `Site` and `Treatment` are assigned.

```
> levels(poplarC$Site) <- c("Moist", "Dry")
> TreatmentLevels <- c("Control", "Fertilizer", "Irrigation", "FertIrriga")
> levels(poplarC$Treatment) <- TreatmentLevels
> str(poplarC$Treatment)

 Factor w/ 4 levels "Control","Fertilizer",..: 1 1 1 1 1 1 1 1 1 1 ...
```

Another way to accomplish the previous labeling that makes clear the assignment of labels to levels is given in R Code 1.34. The reader should make sure that the variable being labeled is a factor before using either the `labels=` or `levels=` argument to assign labels to levels.

R Code 1.34

```
> poplarC$Site <- factor(poplarC$Site, labels = c("Moist", "Dry"))
> str(poplarC$Site)

 Factor w/ 2 levels "Moist","Dry": 1 1 1 1 1 1 1 1 1 2 ...
```

If the argument `levels = c("Moist", "Dry")` is applied to a non-factor variable (as `Site` is in the original `poplar` data frame), the levels of `Site` are converted to `NA` values as seen in R Code 1.35 on the facing page.

R Code 1.35

```
> poplar$Site <- factor(poplar$Site, levels = c("Moist", "Dry"))
> str(poplar$Site)

 Factor w/ 2 levels "Moist","Dry": NA NA NA NA NA NA NA NA NA NA ...
```

The previous examples illustrate assigning labels to levels of a factor. Since the default ordering of the levels of a character factor are alphabetical, one may encounter factors whose levels need to be manipulated. Consider the data frame EPIDURALF from the PASWR2 package that has three levels (`Difficult`, `Easy`, and `Impossible`) for the factor `Ease`. To switch the positions of `Difficult` and `Easy` in the factor level, consider R Code 1.36.

R Code 1.36

```
> library(PASWR2)
> levels(EPIDURALF$ease)  # Default levels (alphabetical)

[1] "Difficult"  "Easy"       "Impossible"

> EPIDURALF$ease <- factor(EPIDURALF$ease, levels = c("Easy",
+     "Difficult", "Impossible"))
> levels(EPIDURALF$ease)  # Correct levels

[1] "Easy"       "Difficult"  "Impossible"

> rm(EPIDURALF)
```

1.11.1 Dealing with NA Values

When working with real data, values are often unavailable because the experiment failed, the subject did not show up, the value was lost, etc. R uses `NA` to denote a missing value or to denote the result of an operation performed on values that contain `NA` values. When performing a computation on an object with `NA` values, one needs to decide what to do with the `NA` values. Many functions (`mean()`, `var()`, `sd()`, `median()`, etc.) have the argument `na.rm=`, which can be set to `TRUE` to ignore the `NA` values and apply the function to the object with the `NA` values removed. Many modeling functions that accept a formula (`t.test()`, `lm()`, `glm()`, etc.) have the argument `na.action=`, which when passed the value `na.omit` will remove any row of the data frame specified in the `data=` argument that has an `NA` value. One should always ask, "Why are the values `NA`?" In certain scenarios, it may be appropriate to impute values for the `NA`s. A simple imputation example is provided in Section 1.18 on page 78. Other times, the user may want to clean up the data so that the `NA` values are removed. By applying the `summary()` function to poplarC, one can see that `Diameter`, `Height`, and `Weight` each have three missing values.

```
> summary(poplarC[, 4:7])  # summary of columns 4-7

    Diameter         Height           Weight       Age
 Min.   :1.030   Min.   : 1.150   Min.   :0.010   3:147
 1st Qu.:3.675   1st Qu.: 5.530   1st Qu.:0.635   4:151
 Median :5.200   Median : 6.950   Median :1.680
 Mean   :4.909   Mean   : 6.969   Mean   :2.117
 3rd Qu.:6.235   3rd Qu.: 8.785   3rd Qu.:3.470
```

```
 Max.   :8.260   Max.   :10.900   Max.   :6.930
 NA's   :3       NA's   :3        NA's   :3
```

Two approaches are presented to remove the missing values for `Diameter`, `Height`, and `Weight`, which, by coincidence, come from the same subjects, which are trees. The first approach (R Code 1.37) uses the function `na.omit()`, while the second approach (R Code 1.38) uses the function `complete.cases()`.

R Code 1.37

```
> dim(poplarC)

[1] 298   7

> myNoMissing <- na.omit(poplarC)
> summary(myNoMissing[, 4:7])  # summary of columns 4-7

    Diameter         Height           Weight      Age
 Min.   :1.030   Min.   : 1.150   Min.   :0.010   3:147
 1st Qu.:3.675   1st Qu.: 5.530   1st Qu.:0.635   4:148
 Median :5.200   Median : 6.950   Median :1.680
 Mean   :4.909   Mean   : 6.969   Mean   :2.117
 3rd Qu.:6.235   3rd Qu.: 8.785   3rd Qu.:3.470
 Max.   :8.260   Max.   :10.900   Max.   :6.930

> dim(myNoMissing)

[1] 295   7
```

The function `na.omit()` removed rows 179, 210, and 218 of the `poplarC` data frame, which had `NA` values in those rows for `Diameter`, `Height`, and `Weight`. The resulting data frame `myNoMissing` maintains the row numbers of the original data frame `poplar` with rows 179, 210, and 218 omitted, which is why the dimension of the `myNoMissing` data frame is 295 by 7, yet the last row is named 298. Compare the original data in `poplarC`, which has rows with `NA` values, to `myNoMissing`, which has the `NA` values removed and row labels maintained for all other rows.

R Code 1.38

```
> poplarC[c(178:180, 209:211, 217:219), ]

     Site Year  Treatment Diameter Height Weight Age
178   Dry    1 FertIrriga     7.57   9.37   5.21   4
179   Dry    1 FertIrriga       NA     NA     NA   4
180   Dry    1 FertIrriga     7.68   9.09   5.12   4
209 Moist    1 FertIrriga     7.28   9.17   4.28   4
210 Moist    1 FertIrriga       NA     NA     NA   4
211 Moist    1 FertIrriga     5.33   8.42   2.36   4
217 Moist    1 FertIrriga     6.58   8.84   3.83   4
218 Moist    1 FertIrriga       NA     NA     NA   4
219 Moist    2    Control     7.71  10.30   5.82   4

> myNoMissing[c(178:179, 208:209, 215:216), ]
```

```
     Site Year  Treatment Diameter Height Weight Age
178   Dry    1 FertIrriga     7.57   9.37   5.21   4
180   Dry    1 FertIrriga     7.68   9.09   5.12   4
209 Moist    1 FertIrriga     7.28   9.17   4.28   4
211 Moist    1 FertIrriga     5.33   8.42   2.36   4
217 Moist    1 FertIrriga     6.58   8.84   3.83   4
219 Moist    2    Control     7.71  10.30   5.82   4
```

The function `complete.cases()` can be applied to a vector, matrix, or data frame and returns a logical vector indicating which cases are complete. In R Code 1.39, the logical vector `complete` is used to extract the rows of `poplarC` that have no missing values.

R Code 1.39

```
> complete <- complete.cases(poplarC)
> myCompleteCases <- poplarC[complete, ]
> dim(myCompleteCases)

[1] 295   7

> summary(myCompleteCases[, 4:7])  # summary of columns 4-7

    Diameter         Height           Weight      Age
 Min.   :1.030   Min.   : 1.150   Min.   :0.010   3:147
 1st Qu.:3.675   1st Qu.: 5.530   1st Qu.:0.635   4:148
 Median :5.200   Median : 6.950   Median :1.680
 Mean   :4.909   Mean   : 6.969   Mean   :2.117
 3rd Qu.:6.235   3rd Qu.: 8.785   3rd Qu.:3.470
 Max.   :8.260   Max.   :10.900   Max.   :6.930

> myCompleteCases[c(178:179, 208:209, 215:216), ]

     Site Year  Treatment Diameter Height Weight Age
178   Dry    1 FertIrriga     7.57   9.37   5.21   4
180   Dry    1 FertIrriga     7.68   9.09   5.12   4
209 Moist    1 FertIrriga     7.28   9.17   4.28   4
211 Moist    1 FertIrriga     5.33   8.42   2.36   4
217 Moist    1 FertIrriga     6.58   8.84   3.83   4
219 Moist    2    Control     7.71  10.30   5.82   4
```

A useful function for testing for the presence of `NA` values in vectors is the function `is.na(x)`. The function returns a logical vector of the same size as `x` that takes on the value `TRUE` if and only if the corresponding element in `x` is `NA`. If `x` is a vector with `NA` values, but only the non-missing values are of interest, the function `!is.na(x)` can be used to extract them as follows.

```
> x <- c(1, 6, 9, 2, NA)
> is.na(x)

[1] FALSE FALSE FALSE FALSE  TRUE

> !is.na(x)

[1]  TRUE  TRUE  TRUE  TRUE FALSE
```

```
> x[!is.na(x)]

[1] 1 6 9 2
```

1.11.2 Creating New Variables in a Data Frame

Two different approaches for adding new variables to a data frame are the `cbind()` and the `within()` functions. The function `cbind()` takes vectors, matrices, data frames, or any combination of vectors, matrices, and data frames as arguments and combines them by columns. The function `rbind()` works in an analogous manner by combining its arguments by rows. Body mass index (`BMI`) is defined as weight (in kilograms) divided by height (in meters) squared. R Code 1.40 creates a new variable, `BMI`, using the information in the `EPIDURALF` data frame. Once the variable is created, it is bound to the `EPIDURALF` data frame using the `cbind()` function and stored in a new data frame named `EPIbmi2`.

R Code 1.40

```
> attach(EPIDURALF)
> BMI = kg/(cm/100)^2  # Creating new variable
> detach(EPIDURALF)
> EPIbmi2 <- cbind(EPIDURALF, BMI)  # Column binding BMI to df
> rm(BMI)  # removing BMI from .GlobalEnv
> EPIbmi2[1:3, -5]  # Show first 3 rows of EPIbmi2 w/o treatment

  doctor  kg  cm      ease oc complications      BMI
1      B 116 172 Difficult  0          None 39.21038
2      C  86 176      Easy  0          None 27.76343
3      B  72 157 Difficult  0          None 29.21011
```

The levels of the variable `Ease` of the `EPIbmi2` data frame are not in ascending order of difficulty, and are subsequently fixed using the `levels()` function.

```
> levels(EPIbmi2$ease)

[1] "Difficult"  "Easy"       "Impossible"

> EPIbmi2$ease <- factor(EPIbmi2$ease, levels = c("Easy",
+     "Difficult", "Impossible"))
> levels(EPIbmi2$ease)

[1] "Easy"       "Difficult"  "Impossible"
```

It should be noted that it is possible, and generally preferable, to create the variable `BMI` directly without using `attach()` by entering

```
> EPIDURALF$BMI <- EPIDURALF$kg/(EPIDURALF$cm/100)^2
> rm(EPIDURALF)
```

Next, the function `within()` is used both to create a new variable `BMI` and to fix the levels of the variable `ease`. The function `within()` can be used to create or modify existing data, including creating new variables with R expressions composed of current objects in the data frame specified by the `data=` argument. Note that the expressions of the `expr=` argument are enclosed in curly braces `{}` with one expression per line.

R Code 1.41

```
> levels(EPIDURALF$ease)

[1] "Difficult"  "Easy"       "Impossible"

> EPIbmi <- within(data = EPIDURALF, expr = {
+   BMI = kg/(cm/100)^2
+   ease = factor(ease, levels = c("Easy", "Difficult", "Impossible"))
+ })
> EPIbmi[1:6, -5]   # Show first 6 rows of EPIbmi w/o treatment

  doctor  kg  cm       ease oc complications      BMI
1      B 116 172  Difficult  0          None 39.21038
2      C  86 176       Easy  0          None 27.76343
3      B  72 157  Difficult  0          None 29.21011
4      B  63 169       Easy  2          None 22.05805
5      B 114 163 Impossible  0          None 42.90715
6      B 121 163  Difficult  3          None 45.54180

> levels(EPIbmi$ease)

[1] "Easy"       "Difficult"  "Impossible"
```

1.11.3 Sorting a Data Frame by One or More of Its Columns

The `sort()` function can be used to sort a single variable in either increasing or decreasing order. Unfortunately, if the user wants to sort a variable in a data frame and have the other variables reflect the new ordering, `sort()` will not work. The function needed to rearrange the values in a data frame to reflect the order of a particular variable or variables in the event of ties is `order()`. Given three variables `x`, `y`, and `z` in a data frame `DF`, the command `order(x)` returns the indices of the sorted values of `x`. Consequently, the data frame `DF` can be sorted by `x` with the command `DF[order(x),]`. In the event of ties, further arguments to `order` can be used to specify how the ties should be broken. Consider how ties are broken with the following numbers in R Code 1.42. To conserve space, the transpose function `t()` is used on the data frame `DF`.

R Code 1.42

```
> x <- c(1, 1, 1, 3, 3, 3, 2, 2, 2)
> y <- c(3, 2, 3, 6, 2, 6, 10, 4, 4)
> z <- c(7, 4, 2, 9, 6, 4, 5, 3, 1)
> DF <- data.frame(x, y, z)
> rm(x, y, z)  # remove x, y, and z from workspace
> t(DF)  # transpose DF

  [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
x    1    1    1    3    3    3    2    2    2
y    3    2    3    6    2    6   10    4    4
z    7    4    2    9    6    4    5    3    1

> with(data = DF, t(DF[order(x, y, z), ]))
```

```
  2 3 1 9 8  7 5 6 4
x 1 1 1 2 2  2 3 3 3
y 2 3 3 4 4 10 2 6 6
z 4 2 7 1 3  5 6 4 9
```

Note that `x` is ordered first as 1, 2, 3. Then, where `x` values are tied, `y` values determine the next ordering. Under the 1 for `x`, `y`'s values appear in 2, 3 order. Under the 2 for `x`, `y`'s values appear in 4, 10 order. Under the 3 for `x`, `y`'s values appear in 2, 6 order. Where `y`'s values are tied, the value for `z` determines the final ordering. Thus, the (`x`, `y`, `z`) triple (1, 3, 2) precedes (1, 3, 7); (2, 4, 1) is before (2, 4, 3); and (3, 6, 4) comes before (3, 6, 9).

Example 1.1 Consider the first 6 rows of the data frame `EPIbmi` (constructed in R Code 1.41 on the preceding page) for the variables `oc`, `ease`, and `BMI`. Sort the data frame `subEPI` first by `oc`, then by `ease`, and finally by `BMI`.

```
> subEPI <- EPIbmi[1:6, c("oc", "ease", "BMI")]
> subEPI

  oc       ease      BMI
1  0  Difficult 39.21038
2  0       Easy 27.76343
3  0  Difficult 29.21011
4  2       Easy 22.05805
5  0 Impossible 42.90715
6  3  Difficult 45.54180
```

Solution: To sort the data frame `subEPI` first by `oc`, then by `ease`, and finally by `BMI`, use the function `order()`. The order for the five subjects that have an `oc` of 0 is first arranged by `ease`, then by `BMI`. Consequently, the order for the two subjects with `oc`s of 0, and level of `ease` of `Easy`, is determined by the `BMI` value.

```
> myO <- order(subEPI$oc, subEPI$ease, subEPI$BMI)
> myO

[1] 2 3 1 5 4 6

> subEPI[myO, ]

  oc       ease      BMI
2  0       Easy 27.76343
3  0  Difficult 29.21011
1  0  Difficult 39.21038
5  0 Impossible 42.90715
4  2       Easy 22.05805
6  3  Difficult 45.54180
```

1.11.4 Merging Data Frames

Related information may be stored in two or more different locations. For example, in a double blind experiment into the efficacy of a new drug claiming to boost the high-density lipoprotein (HDL) of patients, the physician may maintain one set of information, and the

supervising scientist might maintain a separate list indicating who receives the drug and who receives the placebo. In this type of experiment, neither the patient nor the physician knows what type of treatment (drug or placebo) the patient receives; however, a "secret list" is generally maintained by the scientist conducting the experiment indicating who received the drug and who received the placebo. In order to analyze the results, the two data sets will need to be joined based on some common variable. The R function `merge()` is one way to combine data frames from multiple locations. The `merge()` function has named arguments `by.x=` and `by.y=` for situations where the same information is stored in two different data frames under different names. When `merge()` is applied to two data frames without any additional arguments, `merge()` assumes the two data frames have one or more columns with names in common, merges the two data frames, and eliminates any duplicate columns. The default behavior for `merge()` can be changed using the arguments `all=`, `all.x=`, and `all.y=`. Specifying `all = TRUE` will include all rows, `all.x = TRUE` will include all rows from the first data frame, and `all.y = TRUE` will include all rows from the second data frame.

Example 1.2 Consider a fictitious example where a single physician participates in an experiment where she is provided a list indicating she should administer "Treatment One" to patients 1, 5, and 6, and "Treatment Two" to patients 2, 3, and 4. The physician maintains one data base (`DFphy`) with the patient's ID, Gender, and HDL value after finishing some prescribed treatment protocol. The supervising scientist maintains a second data base (`DFsci`) with information on who received the placebo and who received the drug (`secretID`) as well as patient ID. Use the function `merge()` to combine the two data frames.

Solution: Information is first stored in the data frames `DFphy` and `DFsci`, then the two data frames are combined with `merge()`.

```
> DFphy <- data.frame(ID = 1:6, Gender = rep(c("Female", "Male"),
+                     each = 3), HDL = c(39, 42, 22, 27, 29, 45))
> DFphy

  ID Gender HDL
1  1 Female  39
2  2 Female  42
3  3 Female  22
4  4   Male  27
5  5   Male  29
6  6   Male  45

> DFsci <- data.frame(ID = c(2, 4, 3, 5, 1, 6),
+                     secretID = rep(c("Drug", "Placebo"), each = 3))
> DFsci

  ID secretID
1  2     Drug
2  4     Drug
3  3     Drug
4  5  Placebo
5  1  Placebo
6  6  Placebo

> merge(DFphy, DFsci)
```

```
  ID Gender HDL secretID
1  1 Female  39  Placebo
2  2 Female  42     Drug
3  3 Female  22     Drug
4  4   Male  27     Drug
5  5   Male  29  Placebo
6  6   Male  45  Placebo
```

Note that only one column for `ID` appears in the merged result and that the order of the `secretID` has been rearranged to match the variable `ID`.

1.12 Using Logical Operators with Data Frames

Logical operators were first introduced with vectors. Since data frames are collections of equal length vectors having possibly different modes, all of R's logical operators discussed in the context of vectors are still applicable. The data in Table 1.1 that are stored in the data frame `BODYFAT` come from a study reported in the *American Journal of Clinical Nutrition* (Mazess et al., 1984) that investigated a new method for measuring body composition. Suppose one is interested in which subjects have fat percentages less than 25%. Three common approaches that all achieve the same result are `$` prefixing, the `with()` function, and the `attach()` function in combination with a logical statement. The three approaches to answer which subjects have fat percentages less than 25% are illustrated in R Code 1.43.

Table 1.1: Body composition (`BODYFAT`)

n	age	% fat	sex	n	age	% fat	sex
1	23	9.5	M	10	53	34.7	F
2	23	27.9	F	11	53	42.0	F
3	27	7.8	M	12	54	29.1	F
4	27	17.8	M	13	56	32.5	F
5	39	31.4	F	14	57	30.3	F
6	41	25.9	F	15	58	33.0	F
7	45	27.4	M	16	58	33.8	F
8	49	25.2	F	17	60	41.1	F
9	50	31.1	F	18	61	34.5	F

R Code 1.43

```
> head(BODYFAT, n = 3)  # show first 3 rows of BODYFAT

  age  fat sex
1  23  9.5   M
2  23 27.9   F
3  27  7.8   M
```

```
> BODYFAT$fat < 25

 [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> with(data = BODYFAT, fat < 25)

 [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> attach(BODYFAT)
> fat < 25

 [1]  TRUE FALSE  TRUE  TRUE FALSE FALSE FALSE FALSE FALSE FALSE FALSE
[12] FALSE FALSE FALSE FALSE FALSE FALSE FALSE

> detach(BODYFAT)
```

To see the fat percentages for subjects with less than 25% fat, use one of the approaches in R Code 1.44, all of which return the same result. The first three approaches all work with the `fat` vector. The last two approaches extract all rows in the data frame `BODYFAT` where `fat < 25`, for the variable `fat`.

R Code 1.44

```
> BODYFAT$fat[BODYFAT$fat < 25]

[1]  9.5  7.8 17.8

> with(data = BODYFAT, fat[fat < 25])

[1]  9.5  7.8 17.8

> attach(BODYFAT)
> fat[fat < 25]

[1]  9.5  7.8 17.8

> detach(BODYFAT)
> BODYFAT[BODYFAT$fat < 25, "fat"]

[1]  9.5  7.8 17.8

> BODYFAT[BODYFAT$fat < 25, 2]

[1]  9.5  7.8 17.8
```

In R Code 1.44, only the values for the second column of the data frame `BODYFAT`, `"fat"`, were returned by using `BODYFAT[BODYFAT$fat < 25, 2]`. To return a subset of the variables, say `fat` and `sex` in the data frame, one should pass appropriate values to a vector for the columns one wants to extract as illustrated in R Code 1.45.

R Code 1.45

```
> BODYFAT[BODYFAT$fat < 25, c(2, 3)]  # fat < 25 for columns 2 and 3
```

```
   fat sex
1  9.5   M
3  7.8   M
4 17.8   M

> BODYFAT[BODYFAT$fat < 25, c("fat", "sex")] # using names of columns

   fat sex
1  9.5   M
3  7.8   M
4 17.8   M
```

Some may prefer to use the `subset()` function, a convenience function for subsetting. Everything one can do with `subset()` can be done with bracket notation; however, some may find `subset()`'s code more readable. To extract the rows of the column `fat` where `fat < 25` as a vector, use `subset()` as follows.

```
> subset(BODYFAT, select = fat, subset = fat < 25, drop = TRUE)

[1]  9.5  7.8 17.8
```

To extract the rows of the columns `fat` and `sex` where `fat < 25` into a data frame, use `subset()` as seen next.

```
> subset(x = BODYFAT, select = c(fat, sex), subset = fat < 25)

   fat sex
1  9.5   M
3  7.8   M
4 17.8   M
```

Instead of returning only the values of the variable fat that are less than 25%, one might want to see all the values of the other variables where `fat < 25`. Consider the following solutions, which use three different ways to access the information in the data frame. Note that the second argument (`columns`) inside the square brackets after the comma is missing, which causes all variables in the data frame to be returned.

```
> BODYFAT[BODYFAT$fat < 25, ]  # fat < 25 all columns

  age  fat sex
1  23  9.5   M
3  27  7.8   M
4  27 17.8   M

> with(data = BODYFAT, BODYFAT[fat < 25, ])  # fat < 25 all columns

  age  fat sex
1  23  9.5   M
3  27  7.8   M
4  27 17.8   M

> attach(BODYFAT)
> BODYFAT[fat < 25, ]  # fat < 25 all columns
```

```
  age  fat sex
1  23  9.5   M
3  27  7.8   M
4  27 17.8   M

> detach(BODYFAT)
```

The convenience function `subset()` returns the same result as shown next.

```
> subset(x = BODYFAT, subset = fat < 25)

  age  fat sex
1  23  9.5   M
3  27  7.8   M
4  27 17.8   M
```

It is also possible to extract values satisfying more complicated logical conditions. For example, to extract all fat percentages that are less than 25% and different from 7.8, one could enter

```
> with(BODYFAT, fat[fat < 25 & fat != 7.8])

[1]  9.5 17.8
```

To see all the values for the variables in the data frame where fat is less than 25% and different from 7.8, one could enter

```
> with(BODYFAT, BODYFAT[fat < 25 & fat != 7.8, ])

  age  fat sex
1  23  9.5   M
4  27 17.8   M
```

Note that there is only one vector of values for indexing a vector (`fat`) while there are two vectors for indexing a data frame (`BODYFAT`). Another solution is to use the function `subset()`.

```
> subset(x = BODYFAT, subset = fat < 25 & fat != 7.8)

  age  fat sex
1  23  9.5   M
4  27 17.8   M
```

Three additional functions that work with logical objects are `any()`, `all()`, and `which()`. The function `any()` evaluates whether at least one value from a logical statement is true. The function `all()` evaluates whether all values from a logical statement are true. The function `which()` returns the indices for values in which the logical statements are true. R Code 1.46 illustrates the use of `any()`, `all()`, and `which()` on the data frame `BODYFAT`.

R Code 1.46

```
> any(BODYFAT$fat < 10 & BODYFAT$sex == "M")     # Condition TRUE for any?

[1] TRUE

> all(BODYFAT$fat < 10 & BODYFAT$sex == "M")     # Condition TRUE for all?
```