Chapter 2

Brief Introduction to R

The computer examples of this book all use the R program. This chapter is intended for readers who have never encountered R, or for readers who want a refresher on some basic topics. The programming topics discussed here provide necessary building blocks for more complex data manipulation and analysis presented in later chapters. Additional syntax will be presented in future chapters as the need arises.

According to the R FAQ, R is "a system for statistical computation and graphics" (Hornik, 2010). In its basic form, R consists of a `base` package for performing numerical calculations, looping and branching, and various types of statistical analysis. In addition to the `base` package, there are add-on packages written by users to perform all sorts of graphical and statistical analysis. A number of these add-on packages, designed for longitudinal data analysis, will be used in future chapters.

One benefit of R, as opposed to other programs popular in the behavioral sciences, is that it is free. R is licensed under the GNU General Public License to help ensure it remains free (Free Software Foundation, 2007). Overstating the advantages of using a freely available program is difficult. Free software means people with varied resources can use the program, as long as they have an Internet connection. It also implies greater freedom, as one is not constrained by the software choices of the larger entity of which they are a part, such as a business, an academic department, a research lab, etc.

Another advantage of R is that it is syntax-based. This means the user must supply the input to tell the computer what to do, rather than select menu options to accomplish tasks. This certainly is not exclusive to R, but it is nonetheless an advantage. The term *syntax*, or *code*, is used to refer to the input supplied by the user.

The main advantage of a syntax-based program is reproducibility. The analyst can save the computer code as a *script file*, so that all the analyses in one session can be reproduced in a future session. Syntax also facilitates *analysis comprehension*, meaning there is a greater demand on the user to understand the analysis. In most contexts, one must have at least a working knowledge of a topic in order to successfully program in R. The structure of the syntax often provides insights into the nature of the statistical method in question, and how its parameters are estimated.

Because R is a programming language, it is highly flexible and customizable. This means R can perform a wide number of tasks and analyses, with tailoring to the problem at hand. Accomplishing some tasks does require an understanding of some programming skills, an introduction to which appears below.

When one hears complaints about R having "a steep learning curve", perhaps what is being referred to is the fact that knowledge of syntax and functions is necessary to use the program. Many of the functions are part of add-on packages that must be installed. One problem is that R does not provide a self-contained readily accessible list of functions, though an Internet search will reveal lists that have been compiled by users. Consequently, some who are new to R might feel frustrated by the "blank slate" nature of the program.

Two approaches are suggested for facilitating the learning of R. The first is to work through the primer provided in this chapter, and possibly other available primers. The second is to use the help facilities in R, described below in Section 2.10. The help facilities can be used to acquaint the user with available functions and packages to solve specific problems, such as the analysis of longitudinal data.

2.1 Obtaining and Installing R

In order to download and install R, the computer must have a connection to the Internet. The latest version of R (2.12.2 as of this writing) is obtained from the *R Project for Statistical Computing* at `http://www.r-project.org/`. After navigating to the website, click on `CRAN` under `Download, Packages` on the left-hand side of the welcome screen. A server must be selected, called a *CRAN mirror*, in a preferable country of origin (e.g., USA). After selecting a server, the appropriate operating system for the computer must be chosen, either Linux, MacOS, or Windows. For Linux and MacOS, there are directions at the top of the download page for installation. For Windows, the `base` package is downloaded and installed like any other executable file. On Windows machines, the user might need to have administrator privileges to successfully install and use the program. This problem can often be circumvented by installing R outside of the default `Program Files` directory (e.g., install in `C:\R`).

A FAQ for each operating system is available on the website, and should be consulted in the event there are problems with downloading or installing the program. User-written help documents and videos can be obtained with a judicious Internet search.

2.2 Functions and Packages

The `base` package has an impressive variety of functions for manipulating, analyzing, and graphing data. For more specialized analysis, such as that involving longitudinal data, it is often advantageous to use one of the user contributed add-on packages. These packages typically have specialized functions tailored for specific topics. Many of the add-on packages are written by world leading authorities on the topic of interest.

Throughout this book, names of packages and functions appear in `typewriter font`. Functions will be written with trailing parentheses, as in `mean()`, to denote the fact that *arguments* must be supplied to the functions. The arguments constitute pieces of information that influence the workings of the function. For instance, many functions require an argument naming the variable to be analyzed and options for treating missing data. The following is an example of the `mean()` function with two arguments (arg. 1 and arg. 2) that will be explained below.

$$\overbrace{\texttt{mean(}\underbrace{\texttt{ x }}_{\text{arg. 1}},\ \underbrace{\texttt{na.rm} = \texttt{TRUE}}_{\text{arg. 2}}\texttt{)}}^{\text{function}}$$

Add-on packages, beyond the `base` package, must be installed before they can be used. The easiest method of installing is within R. After invoking R, an add-on package can be installed by using menu options. For Windows computers, this is accomplished with the `Packages` menu option. Alternatively, the `install.packages()` function can be used at the prompt or from a script file. The name of the package to be installed must be supplied in quotes, along with the quoted name of the repository or remote server that houses the package. A list of repositories is provided on the *R Project for Statistical Computing* website.

As an example, consider installation of the add-on package `ggplot2`, useful for graphing longitudinal data. Assume the `base` package has been successfully installed, and the R program has been invoked (started). The following syntax installs the `ggplot2` package from the repository at the University of California at Los Angeles (UCLA):

```
install.packages("ggplot2", repos = "http://cran.stat.ucla.edu/")
```

To be perfectly clear, the above syntax is typed at the prompt, `>`, in the R console window, which is the window that appears when R is first invoked. After typing the syntax, the Enter key is pressed to execute it. Alternatively, the syntax can be executed from a script file that is explained in a moment. After submitting the syntax, the console window will show some relatively cryptic messages, but the last line should indicate successful installation.

In addition to the `ggplot2` package, extensive use is made of `lme4` and `AICmodavg` in the following chapters. Additional packages will be mentioned as needed. Many primary packages link to, or depend on, other ancillary packages. These ancillary packages are usually automatically installed along with the primary package. For example, when the `ggplot2` package is installed, the `plyr` package for data manipulation and analysis is also automatically installed. To ensure that dependent packages are always installed, the additional argument `dependencies = TRUE` should be included in the `install.packages()` function.

Once a package is installed, its functions are not automatically made available. The `library()` or `require()` function must be used to load the package. If you want to use the functions in the `ggplot2` package, for example, type `require(ggplot2)` at the prompt or execute the syntax from a script file. Ancillary packages are also made available when a primary package is loaded. The `require(ggplot2)` not only loads `ggplot2`, but also loads `plyr` and other related packages.

2.3 Essential Syntax

Every syntax-based computer program has its own conventions. The following sections outline concepts and syntax that are essential building blocks for the data manipulation and analysis covered later in this book.

2.3.1 Prompt versus Script Files

One method of executing syntax is to type it at the prompt in the R console window and press the Enter key. Syntax can also be typed in a *script file* and executed in part or whole. R provides a script editor for creating script files that is accessed by the menu options. There are a whole host of free third-party script editors that are also available.

The main advantage of the script file is the contents can be saved. The script file is a record of the analyses you perform and the file can be opened in future R sessions. The script file should be periodically saved using the appropriate menu options or key strokes. The script file is saved with the extension `*.r` or `*.R`.

The computer examples in this book depict the syntax as being typed and executed from the prompt in the console window. However, this is done for clarity of presentation, and it is suggested that a script file be routinely used.

2.3.2 Input and Output Appearance in this Book

In this book, R syntax and output appear in `typewriter font` that is slightly smaller than the narrative text. In addition, input and output are shaded in gray. Syntax (input) is always preceded by the symbol `>`, which is the R prompt in the console window. When syntax runs over one line, a plus sign (`+`) appears at the beginning of any continuation lines. If the reader is copying syntax from the book, the plus sign should not be submitted as part of the syntax. Output is always $\boxed{\text{boxed}}$ to distinguish it from syntax. Depending on the nature of the output, it might be indexed by numbers appearing in single brackets, for example, `[1]`, or double brackets, `[[2]]`. The details of indexing are discussed later in this chapter.

In certain cases, it is desirable to discuss specific parts of multi-line syntax or output. To facilitate this, line numbers will sometimes appear to the left of the syntax or output box, so that reference can be made to specific material. The following is an example, but do not worry about the meaning of the syntax for the time being.

```
> summary(c(10, 5, 2))
```

```
1    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
2   2.000   3.500   5.000   5.667   7.500  10.000
```

Line 1 contains the headings of the columns. Line 2 contains the numerical output that will be explained later. The reader familiar with elementary statistics can probably determine the meaning of the statistics by the column headings (e.g., `Min.` is the minimum value).

2.3.3 Quitting R

The R session is terminated by typing `q()` at the command prompt, or running the syntax from the script file, or selecting the appropriate menu options (e.g., `File, Exit` on Windows machines). When quitting, there is a prompt to save the workspace image. The workspace image consists of the objects created in the session (see below), but **not** the syntax. The workspace is saved in a file that is only readable by `R`, and has the extension `*.rdata` or `*.Rdata`. Typically, important things like data sets and output are saved within the session, so the workspace image need not be saved when exiting.

2.3.4 Terminating a Process

Sometimes a syntax mistake is made, and a process is initiated that is unintended on the part of the user. Other times a closing parenthesis or curly bracket (}) is omitted and the user cannot return to the console prompt. In these situations, the Esc key (escape key) can be used on a Windows machine to terminate a process. This returns the cursor to the console prompt. The menu option `Misc` can also be used.

2.3.5 Basic Calculations

A basic use of `R` is as a calculator. Using commands for addition (`+`), multiplication (`*`), division (`/`), exponents (`^`), etc., all the typical operations available on hand-held calculators can be performed. Suppose you want to carry out the calculations in the following equation,

$$(2+3)+(2\cdot 3)-\left(\frac{6}{3}\right)-3^2=0. \tag{2.3.1}$$

The syntax for the operations is shown below. When the syntax is executed, the result is printed to the computer screen, which is the same as saying it is printed to the `R` console window.

```
> (2 + 3) + (2 * 3) - (6 / 3) - 3 ^ 2
```

```
[1] 0
```

The input and output above are similar to what appears when the syntax is executed from the `R` prompt. As shown above, syntax is preceded by the `>` symbol, and each line of syntax has its output immediately below, appearing in a box. Boxing is used to facilitate understanding in this text, but it is not used in the `R` program. The `[1]` preceding each output line is for indexing, conveying that there is only one line of output.

Basic calculations are often used within more complex operations. An advantage of `R` is that several things can be done "on the fly" by embedding simpler operations within more complex ones. Examples appear in later sections.

2.3.6 Objects

`R` is an object-oriented program. Objects are named storage entities to which things like numbers, words, or data sets are *assigned*. The assignment of things to objects requires use of the assignment operator, `<-`. The assignment operator is formed with the `<` and `-` keys. The required form is,

$$\text{object name} \leftarrow \text{thing to be assigned.}$$

An example will help clarify the assignment concept. Suppose the result of simple addition is assigned to the object `myresult`.

```
> myresult ← 2 + 3
> myresult
```

```
[1] 5
```

The object stores the result of the operation, and its contents are displayed by typing the object name. This is an *implicit print*. An *explicit print* can be performed by using the `print()` function, as in `print(myresult)`. Explicit print is sometimes desirable, as additional arguments can be used to alter the output.

One of the powerful features of `R` is that after an object is created, it can be used in subsequent operations. As an illustration, consider the following.

```
> myresult * 2
```

```
[1] 10
```

In the example, `myresult` consists of a number, and it can be manipulated in the same way as using the number directly. In most of the analyses presented in future chapters, extensive use of objects is made.

Object names are arbitrary, but there are some conventions that should be followed when naming. Names cannot begin with a number, and should not have spaces. A period rather than an underscore (or other character) should be used for clarity, for instance, `my.result`. Objects should not have the same names as functions (see below), so names like `mean` or `sd` are to be avoided. In addition, **`R` is case-sensitive**. You cannot type `Myresult` to display `myresult`, as the two names are interpreted as different objects.

Up to this point, objects have been proxies for numbers. The object-oriented approach is more impressive when one considers that an object can consist of complex things, such as a data set, or the output from a statistical analysis. An example of storing a data set as an object is presented in Section 2.5.3. In Section 2.9, the output of a statistical analysis is saved as an object.

2.3.7 Concatenation

A group or list of numbers, known as a vector, is assigned to an object using the concatenation function, `c()`. The following syntax assigns a vector of numbers to the object `mydat`.

```
> mydat ← c(2, 3, 1, 0, 3, 4)
> mydat
```

```
[1] 2 3 1 0 3 4
```

The commas are necessary separators for the numbers in the syntax, but the spaces are optional. Because the values in the vector are numbers, `mydat` is a *numeric vector*. The `mydat` object can be used in similar operations as an object consisting of a single number.

It is important to understand how `R` interprets the syntax when the object is a vector. Suppose `mydat` is multiplied by 2.

```
> mydat * 2
```

```
[1] 4 6 2 0 6 8
```

Comparison with the previous output shows that the original vector is multiplied by 2 on an element-by-element basis. That is, each element of `mydat` is multiplied by 2. Functions are available for other types of operations on vectors that are not element-by-element. Some of these are shown in the Appendix, where an introduction to matrix algebra is presented. Additional functions are introduced as needed in the following chapters.

The concatenation operator can be used with non-numeric data. For example, a vector of names rather than numbers is created below.

```
> mynames ← c("Carlos", "Marta", "Salome", "Philip")
> mynames
```

```
[1] "Carlos" "Marta"  "Salome" "Philip"
```

The quotes are necessary in `c()` for non-numeric data, and the resulting object is a *character vector* or *string vector*. In printing the character vector, the quotes are removed by using an explicit print with the logical argument `quote=FALSE`, as in `print(mynames, quote=FALSE)`.

`R` will return an error message if non-permissible operations are attempted with character vectors. For example, the following will produce an error message.

```
> mynames * 2
```

There are permissible operations with character vectors. As an example, two character vectors are concatenated to produce a new character vector using the `paste()` function.

```
> paste("grade.", as.character(5:8), sep = "")
```

```
[1] "grade.5" "grade.6" "grade.7" "grade.8"
```

As shown in the output, the `as.character()` function converts the numbers $5, 6, 7, 8$ to characters, and then concatenates each with `grade`. The `sep=` argument sets no separation space between the concatenated elements (the default is a single space).

2.3.8 Statistical Functions

The `base` package has a number of useful statistical functions. The statistical functions are usually used with vector objects, but they can also be used with other types of objects. Below are examples of some common statistical functions and the output they produce. Various statistics are computed for the `mydat` vector from the previous section.

```
> mydat                                   # Display the data.
```

```
[1] 2 3 1 0 3 4
```

```
> mean(mydat)                             # Mean.
```

```
[1] 2.166667
```

```
> sd(mydat)                               # Standard deviation.
```

```
[1] 1.47196
```

```
> summary(mydat)                          # Descriptive statistics.
```

```
  Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
 0.000   1.250   2.500   2.167   3.000   4.000
```

```
> min(mydat); max(mydat)                  # Minimum and maximum values.
```

```
[1] 0
```

```
[1] 4
```

The pound sign (`#`) is used for comments, allowing the input program to be annotated for increased comprehension. Everything following the pound sign to the end of the line will be ignored by the program. The last line of syntax uses a semi-colon (`;`) to separate two functions. This convention must be followed if more than one function is used on a single line. As shown above, the output is stacked when the semi-colon is used.

A number of additional statistical functions will be introduced in the following chapters. Many of these functions are specific to the package designed for particular purposes.

2.4 Data Types

There are different data types in `R`, and vectors of data are characterized by the data they contain. Data types are important to consider, as they have implications for data analysis. Two types have already been encountered, numeric data and character data.

A third data type is a *factor*. A factor variable represents a categorical variable, or grouping variable, that uses numbers to represent its categories or levels. Factor variables are created with the `factor()` function. One method of creating a factor variable is to first create a numeric vector, and then associate character descriptors with each number. It is common for the factor vector to have few distinct values, perhaps two or three.

Suppose the goal is to create a factor variable of gender, with coding of male and female. First, a vector of 0s and 1s is created, and then 0 is associated with "Male" and 1 is associated with "Female". The following syntax accomplishes the task.

```
> myfac0 ← c(0, 1, 1, 0)
> myfac ← factor(myfac0, labels = c("Male", "Female"))
> myfac
```

```
1 [1] Male   Female Female Male
2 Levels: Male Female
```

Line 1 of the output contains an index, `[1]`, and then a listing of the factor vector contents. Line 2 of the output provides an exhaustive list of the category labels. In this instance, the exhaustive list is not needed, as the label categories are readily apparent from the listing of the factor vector contents. In other situations, the exhaustive list is helpful in understanding the details of the factor vector.

A function helpful for understanding output objects is the structure function, `str()`. The structure function provides a partial listing of vector contents and provides information about data type. Suppose that `str()` is used with `myfac`.

```
> str(myfac)
```

```
 Factor w/ 2 levels "Male","Female": 1 2 2 1
```

The output indicates `myfac` is a factor variable with two levels, and the levels are coded as “Male” and “Female”. The numbers on the right index the levels and are in the same order as the original values. The numeric levels are produced in this case by adding 1 to the original values of 0 and 1, yielding 1 and 2, respectively. Factor vectors are very useful when one wants to include a categorical variable in an analysis. Examples are shown in future chapters.

The `str()` function can be used with almost any output object. Consider its use with a numeric vector and a character vector.

```
> mynum ← c(1.5, 10, 8.467, 8)
> str(mynum)
```

```
 num [1:4] 1.5 10 8.47 8
```

```
> mychar ← c("Carlos", "Marta", "Salome", "Philip")
> str(mychar)
```

```
 chr [1:4] "Carlos" "Marta" "Salome" "Philip"
```

In the output, `num` stands for numeric and `chr` stands for character. The bracketed numbers, `[1:4]`, refer to there being 1 through 4 elements in the vectors; more on this later.

2.4.1 Missing Values

An important consideration with all data types is the coding of missing elements, or missing values. In `R`, missing values are represented as `NA`, “not available.” `NA` is used when an observation is not available, but a place holder is desired anyway. Consider an example of a numeric vector with the second element missing.

```
> mynum2 ← c(1.5, NA, 8.467, 8)
> mynum2
```

```
[1] 1.500    NA 8.467 8.000
```

In the concatenation function, `NA` is not quoted. The output shows the second element is not available, but a space (or location) is retained for it anyway.

In the `NA` example, why not skip over the missing value and simply assign `c(1.5, 8.467, 8)`? The answer is that it is often desirable to reserve a space for observations that were intended, but not realized. Suppose the intention is to obtain scores from four subjects, but the second subject’s score cannot not be ascertained. For bookkeeping purposes, it is desirable to index the fact that there was an attempt to obtain scores from four subjects. The bookkeeping is especially important when observations are collected on many variables. Certain subjects may have missing values on some variables, but not others. It is confusing to have the data vectors contract and expand based on the missing data. Including missing data values allows a clear matching of the subjects with their data.

Missing values present issues for the use of R functions. All functions have a default method of treating missing data that can be changed with optional arguments. The default method for a function is found on the help page of each function, as described later.

Some functions, such as `summary()`, will ignore missing data by default and compute descriptive statistics on the non-missing values. Values other than `NA` are referred to as *available values*. In what follows, the `summary()` function is used on the numeric vector with a missing value.

```
> mynum2
```

```
[1] 1.500    NA 8.467 8.000
```

```
> summary(mynum2)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
  1.500   4.750   8.000   5.989   8.234   8.467   1.000
```

As the output shows, descriptive statistics are computed on the available values. The `summary()` function counts and displays the number of missing values in the `NA's` column.

Other functions, such as `mean()` and `sd()`, do not use available values by default. Consider what happens when `mean()` is applied to the vector with a missing value.

```
> mean(mynum2)
```

```
[1] NA
```

The result is a missing value. Though this default might appear inconvenient at first, it is useful for alerting the user to the presence of missing data. This is not crucial in the small example, but it can be with larger data vectors.

In order to have functions like `mean()` ignore missing values, one must include the logical argument `na.rm=TRUE`. A logical argument takes the value of `TRUE` or `FALSE`. The logical argument tells the function to remove the missing values and then compute the mean. Consider what happens when the optional syntax is used.

```
> mean(mynum2, na.rm = TRUE)
```

```
[1] 5.989
```

Now the mean of the available values is computed. In the chapters to follow, additional details regarding missing values and their treatment are discussed.

2.5 Matrices, Data Frames, and Lists

There are three main formats, or *data structures*, for the arrangement of data in R. These are a *matrix*, a *data frame*, and a *list*. A summary of the main features is shown in Table 2.1.

A matrix is a rectangular array for only one data type. A soft introduction to matrices is provided in the Appendix. A vector is a special case of a matrix that has only one column or one row. A data frame is similar to a matrix, but different columns (vectors) can have different data types. However, you cannot mix types of data in the same column. The use of `NA` in a numeric vector may appear to violate the same-column convention, but `NA` is **not** a character string in R.

An important characteristic of both a matrix and a data frame is that the columns are of the same length. A list is similar to a data frame allowing different vectors to have different types of data, but the vectors need not be of the same length.

2.5.1 Vector

As seen above, a vector is constructed with the concatenation function. Suppose that the numeric and character vectors previously considered are once again created.

```
> mynum <- c(1.5, 10, 8.467, 8)
> mychar <- c("Carlos", "Marta", "Salome", "Philip")
```

These vectors will be considered for calculations in the sections below.

2.5.2 Matrix

A matrix can be made from the two vectors, `mynum` and `mychar`. As indicated in Table 2.1, a matrix must have columns (vectors) of equal length, and `mynum` and `mychar` meet this requirement. In addition, a matrix can only have one type of data. This is a problem because one vector is numeric and the other is character. In such situations, R will coerce the data to be of one type.

The `matrix()` function is used to create a matrix. The concatenation function is used to list the two vectors that will be the columns of the `mymat` matrix. The argument `ncol=` specifies the number of columns of the matrix. The number of rows is determined from the `ncol=` argument and the data.

```
> mymat ← matrix(c(mynum, mychar), ncol = 2)          # Matrix with 2 columns.
> mymat
```

```
     [,1]    [,2]
[1,] "1.5"   "Carlos"
[2,] "10"    "Marta"
[3,] "8.467" "Salome"
[4,] "8"     "Philip"
```

The output shows that each element of `mymat` is quoted. Quotes denote character data, illustrating that R forces the numeric vector to be a character vector in the matrix. The column and row brackets (e.g., `[1,]`) are used for indexing to be discussed in a moment. For now, it should be evident the matrix consists of two columns, each having four rows.

To gain further insight, the `str()` function is used with the matrix object.

```
> str(mymat)
```

```
 chr [1:4, 1:2] "1.5" "10" "8.467" "8" "Carlos" "Marta" ...
```

The output reveals that `mymat` is a character matrix (`char`). A partial listing of the matrix elements is displayed, beginning with the first column.

2.5.3 Data Frame

A data frame is created using the `data.frame()` function. A data frame is also created with `read.table()`, which is used to read an external text file into R. The reading in of data is discussed in the next chapter.

Table 2.1 shows that a data frame has columns with equal numbers of rows, but the data types can differ. Consider a data frame created from the numeric and character vectors. Column labels, or variable names, are specified by using the equal sign (=).

```
> mydat ← data.frame(Char = mychar, Num = mynum)              # Data frame.
> mydat
```

```
    Char    Num
1 Carlos  1.500
2  Marta 10.000
3 Salome  8.467
4 Philip  8.000
```

As the syntax shows, `data.frame()` does not require the use of `c()`. The vector objects to be included are listed, separated by a comma.

The output illustrates some useful characteristics of data frames. The character vector and the numeric vector are displayed without quotes, indicating that different data types are associated with different columns. The headings of the columns are known as *column names*, which can be changed. The numbers to the extreme left of the vectors are *row names*, which also can be changed.

Consider the use of the `str()` function with the data frame object.

```
> str(mydat)
```

```
'data.frame':   4 obs. of  2 variables:
 $ Char: Factor w/ 4 levels "Carlos","Marta",..: 1 2 4 3
 $ Num : num  1.5 10 8.47 8
```

Line 1 of the output shows `mydat` is a data frame with four rows, and two columns (two variables). Line 2 lists the first column name after the dollar sign (`$`), and reveals it is a factor variable. Both the category labels and the level numbers are listed. The `data.frame()` function converts a character vector to a factor vector by default. Character vectors are retained as such by using the logical argument `stringsAsFactors=FALSE`. Line 3 shows the second column is numeric (`num`) and lists its values.

2.5.4 List

A list is created using the `list()` function. Table 2.1 shows that a list can have different data types and different column lengths. In order to illustrate different column lengths, a list with three elements is created, the last element being a numeric vector with one row shorter than the others.

```
> mylist ← list(Char = mychar, Num4 = mynum, Num3 = c(1, 2, 3))         # List.
> mylist
```

```
$Char
[1] "Carlos" "Marta"  "Salome" "Philip"

$Num4
[1]  1.500 10.000  8.467  8.000

$Num3
[1] 1 2 3
```

The output for the `mylist` list object depicts the vectors in rows rather than columns. The character vector has quoted elements, whereas the numeric vectors do not, illustrating that mixed data types are allowed. The third vector is one element shorter than the first two, but all the vectors are displayed. The bracketing (`[1]`) is indexing, which is discussed in the next section.

Further insight is obtained by using the `str()` function on the list object.

```
> str(mylist)
```

```
List of 3
 $ Char: chr [1:4] "Carlos" "Marta" "Salome" "Philip"
 $ Num4: num [1:4] 1.5 10 8.47 8
 $ Num3: num [1:3] 1 2 3
```

Line 1 indicates the `mylist` object is a list with three elements. Line 2 shows the first vector is a character vector (`char`). Lines 3-4 reveal the remaining two vectors are numeric and are of different lengths.

Table 2.1: Features of R data structures.

Structure	Data Type	Indexing	Column Lengths	Function
Vector	Single	`[row]` or `[column]`	Not Applicable	`c()`, `matrix()`
Matrix	Single	`[row, column]`	Equal	`matrix()`
Data Frame	Mixed	`[row, column]`	Equal	`data.frame()`, `read.table()`
List	Mixed	`[[vector]][row]`†	Unequal	`list()`

†Assumes elements are vectors (see text).

Working with Data Frames

The data frame will be the primary data structure in future chapters, so it is beneficial to mention some additional characteristics. First, the column names of a data frame are changed using the `colnames()` function. Second, a column (vector) is accessed or extracted by using the data frame object name followed by a dollar sign (`$`) and the column name. Consider an illustration.

```
> colnames(mydat)                        # Shows the column names.
```

```
[1] "Char" "Num"
```

```
> colnames(mydat) ← c("Name", "Score")    # Assigns new column names.
> mydat                                   # Implicit print.
```

```
    Name  Score
1 Carlos  1.500
2  Marta 10.000
3 Salome  8.467
4 Philip  8.000
```

```
> mydat$Name                              # Print first column (variable).
```

```
[1] Carlos Marta  Salome Philip
Levels: Carlos Marta Philip Salome
```

```
> mydat$Score                             # Print second column (variable).
```

```
[1]  1.500 10.000  8.467  8.000
```

Small data frames with a limited number of rows and columns are easily created with the concatenation function, as shown above. Larger data frames are usually created by reading in a text file of the data set with the `read.table()` function. Discussion of `read.table()` is deferred to the next chapter.

An alternative to the dollar sign syntax for accessing variables is `attach()` and `detach()`. By including the data frame name in `attach()`, the variables of the data frame are made available directly by their names. The direct naming is ended by including the data frame name in `detach()`. Consider the following.

```
> attach(mydat)
> mean(Score)
```

```
[1] 6.99175
```

```
> sd(Score)
```

```
[1] 3.759513
```

```
> detach(mydat)
```

Since `Score` is a variable in the `mydat` data frame, it is accessed directly after executing `attach(mydat)`. The access is ended by executing `detach(mydat)`. Using `mean(Score)` outside of `attach()` and `detach()` returns an error.

Any number of procedures can be performed between `attach()` and `detach()`, and above two were shown for brevity. It is always good to use `detach()` when finished with the data frame. This avoids the potential problem of similarly named variables in one data frame replacing (masking) variables in another data frame.

Another alternative is the `with()` function, which is similar to `attach()` and `detach()`, but is used a single time. In the syntax below, `with()` is used to allow the variable names to appear in `mean()`.

```
> with(mydat, mean(Score))
```

```
[1] 6.99175
```

```
> with(mydat, c(mean(Score), sd(Score)))
```

```
[1] 6.991750 3.759513
```

The second line of syntax uses `c()` to perform two tasks.

2.6 Indexing

Accessing and extracting individual elements of a vector, matrix, data frame, or list is accomplished using the indexing conventions of R. See Table 2.1 for a summary of syntax conventions.

2.6.1 Matrix and Data Frame

For a matrix and a data frame, each element has a location defined by the row and column where it resides. The row and column address is denoted by square brackets separated by a comma, `[ , ]`. For example, `mymat[1,2]` refers to the element in the first row and second column of the matrix object. Listing the object name and bracket indexes implicitly prints the element to the computer screen. Here are some examples.

```
> mymat                                      # Display all matrix elements.
```

```
     [,1]    [,2]
[1,] "1.5"   "Carlos"
[2,] "10"    "Marta"
[3,] "8.467" "Salome"
[4,] "8"     "Philip"
```

```
> mymat[1,2]                                 # First row, second column.
```

```
[1] "Carlos"
```

```
> mydat                                      # Display all data frame elements.
```

```
    Name  Score
1 Carlos  1.500
2  Marta 10.000
3 Salome  8.467
4 Philip  8.000
```

```
> mydat[2,2]                                 # Second row, second column.
```

```
[1] 10
```

Several row and/or column elements can be indexed in various ways. Contiguous elements are indexed by a colon (:), and the concatenation function is used for non-contiguous elements. An entire row or column is denoted by a blank. Examples using these conventions are the following.

```
> mymat[1:2,2]                               # First two rows of the second column.
```

```
[1] "Carlos" "Marta"
```

```
> mydat[1:2,1:2]                             # First two rows, first two columns.
```

```
    Name Score
1 Carlos   1.5
2  Marta  10.0
```

```
> mymat[c(1,3),1]                            # First and third rows of the first column.
```

```
[1] "1.5"   "8.467"
```

```
> mydat[ ,1]                                 # All rows of first column.
```

```
[1] Carlos Marta  Salome Philip
Levels: Carlos Marta Philip Salome
```

```
> mydat[1, ]                                 # All columns of first row.
```

```
    Name Score
1 Carlos   1.5
```

2.6.2 Vector

A vector is a matrix with only one column or one row. As a consequence, vector objects have only a single index. Elements of a vector are referred to by a single entry in brackets. Consider the following.

```
> mynum ← c(1.5, 10, 8.467, 8)
> mynum[1]                          # First element.
```

```
[1] 1.5
```

```
> mynum[3]                          # Third element.
```

```
[1] 8.467
```

The vector index is often used in output. Each output example above consists of a single row. The output indexing is particularly helpful when output wraps onto multiple lines.

2.6.3 List

Indexing for lists requires double square brackets, `[[ ]]`. The double brackets are used to access specific vectors. Single brackets are used in conjunction with double brackets to access elements of specific vectors. Consider these examples.

```
> mylist                            # Print all list elements.
```

```
$Char
[1] "Carlos" "Marta"  "Salome" "Philip"

$Num4
[1]  1.500 10.000  8.467  8.000

$Num3
[1] 1 2 3
```

```
> mylist[[2]]                       # Second vector.
```

```
[1]  1.500 10.000  8.467  8.000
```

```
> mylist[[2]][2]                    # Second element of second vector.
```

```
[1] 10
```

```
> mylist[[3]]                       # Third vector.
```

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

```
> mylist[[3]][1]                    # First element of third vector.
```

```
[1] 1
```

In the examples here, the list elements are vectors, and the indexing represented in Table 2.1 applies. However, lists are general, and the individual elements can be almost anything like matrices, data frames, and statistical output. In these more complex situations, the double bracket notation can be used to access a list element, but the single bracketing might require both a row and column, or single bracketing might not work at all.

2.6.4 Sorting

Indexing can be used to sort the rows of data. As discussed in future chapters, it is often desirable to sort the rows of a data frame by the subject identification number.

Sorting of rows is accomplished by using the `order()` function in the row index of a data frame. The sort variable must be supplied to `order()`. Consider sorting the `mydat` data frame by `Score`.

```
> mydat                                 # Unsorted data frame.
```

```
    Name  Score
1 Carlos  1.500
2  Marta 10.000
3 Salome  8.467
4 Philip  8.000
```

```
> mydat2 ← mydat[order(mydat$Score), ]    # Sort data frame by Score.
> mydat2                                  # Sorted data frame.
```

```
    Name  Score
1 Carlos  1.500
4 Philip  8.000
3 Salome  8.467
2  Marta 10.000
```

The `order()` function appears in the row index, and the column index is left blank, indicating the intent to sort all the columns (variables). The sort variable, `Score`, must be attached to its data frame, `mydat`, using the dollar sign, so that R properly recognizes it. The sorted data is saved in a new data frame, `mydat2`. As shown here, it is recommended that data frame manipulations be saved to a new object in case of error.

2.6.5 Recoding

Indexes can be used to recode specific elements. Suppose the goal is to recode the second element in the second column of the `mydat2` object to `NA`. The following syntax accomplishes this goal.

```
> mydat2[2,2] ← NA                        # Recode a value to NA.
> mydat2
```

```
    Name  Score
1 Carlos  1.500
4 Philip     NA
3 Salome  8.467
2  Marta 10.000
```

2.6.6 Saving Objects

Objects are saved to file with the `save()` function. A single object or multiple objects can be saved in a single file. The saved file has the extension `*.rdata` or `*.Rdata`. The objects to be saved must be supplied and the name and location of the Rdata file. Consider saving the `mynum` and `mydat` objects to a file on a Windows computer.

```
> save(mynum, mydat, file = "C:/Mine/myfile.Rdata")
```

The `mynum` vector and the `mydat` data frame are both saved to `myfile.Rdata`. Note the use of the forward slash (`/`), which can also be replaced by a double back slash (`\\`). Rdata files are only recognized by R. Text files can be written with the `write.table()` function discussed in the next chapter.

2.6.7 Loading and Listing Objects

A saved Rdata file is loaded into R using the `load()` function. The quoted file name and location must be supplied. Once the objects are loaded into computer memory, a listing of available objects is obtained with `ls()` or `objects()`. Any object can be deleted from memory – but not from the Rdata file – using `rm()`. To delete objects from a Rdata file, a new Rdata file is saved omitting the desired object(s).

In the syntax below, all objects from memory are deleted. Then the `myfile.Rdata` file saved above is loaded. After loading the file, the objects in active computer memory are listed. Finally, it is shown how an individual object is deleted from computer memory.

```
> rm(list = ls())                            # Delete all objects from memory.
> ls()                                       # No objects available.
```

```
character(0)
```

```
> load(file="C:/Mine/myfile.Rdata")
```

```
> objects()
```

```
[1] "mychar" "mydat"  "mynum"
```

```
> rm(mynum)
> objects()
```

```
[1] "mychar" "mydat"
```

2.7 User-Defined Functions

For situations encountered later in this book, it is desirable to define functions that are not part of the standard set in the `base` package or the add-on packages. Such functions are known as *user-defined functions*. User-defined functions are commonly used to handle special tasks, such as bootstrap simulations in LMER analysis; see the optional sections of Chapters 7 and 8.

User-defined functions are created with `function()`. The contents of `function()` are assigned to an object, and then the object is used in a similar manner as the other functions discussed in this chapter.

As an illustration, consider defining a function for computing the sample mean. There is no need to write a function for the sample mean, as `mean()` already exists. But this task affords the opportunity to compare and check the results of the user-defined function against an existing function from the `base` package.

Suppose the `mynum` numerical vector is created again, and a function is written to compute the mean. The goal is to compute the sum of the elements of `mynum` and divide by the total number of elements, or the length of the vector. The `sum()` and `length()` functions are used along with division, `/`, to accomplish the goal. Consider the following.

```
> mynum ← c(1.5, 10, 8.467, 8)
> mymean ← function() {sum(mynum) / length(mynum)}
```

The operations are enclosed in curly brackets, `{ }`, but this is optional. The parentheses of `function()` must be included, though they are empty in this example. The syntax indicates the function will compute the sum of `mynum` and divide by its length.

The above syntax defines the `mymean()` function. Now the function is used and checked against the result of `mean()`.

```
> mymean()
```

```
[1] 6.99175
```

```
> mean(mynum)
```

```
[1] 6.99175
```

As the output shows, the user-defined `mymean()` function yields the same result as the `mean()` function.

In its current state, `mymean()` will only work on the `mynum` vector. To apply the function to a vector other than `mynum`, a variable must be specified in the definition. Any character string will work as a variable, and here the simple alternative of `x` is used. By incorporating `x` into the syntax in the following manner, it is possible to apply `mymean()` to any vector.

```
> mymean <- function(x) {sum(x) / length(x)}
```

Now an argument must be passed to `mymean()` to replace `x` in the function definition. In this case, the argument is the name of the vector for which the mean is to be computed. This can be made explicit by using the optional argument `x=`.

```
> mymean(x = mynum)
```

```
[1] 6.99175
```

```
> mynum2 <- c(10, 8, 2, 0)
> mymean(mynum2)
```

```
[1] 5
```

As shown in the output, `mymean()` can now be used with any numeric vector. Missing data treatments are not programmed in `mymean()`, which is a reason to use `mean()`. In general, an existing function is preferred to a user-defined function. However, in later chapters, operations are encountered that are not encompassed in any existing functions. In such cases, a user-defined function is valuable.

2.8 Repetitive Operations

In later chapters, statistical procedures known as bootstrap methods will be discussed. Bootstrap methods require that the same statistical operations to be repeated a large number of times. Repetitive operations can be carried out in a number of ways, but two methods are particularly useful. The first uses the `rdply()` function from the `plyr` package, written by Hadley Wickham (Wickham, 2009a). The second uses a `for()` loop, which is the `R` version of a loop structure common in general computer programming.

It is easiest to use `rdply()` in conjunction with a user-defined function, which is illustrated below. With a `for()` loop, the user-defined function is incorporated into the programming structure. For the novice programmer, `rdply()` is easier to use than a `for()` loop, as the conventions of the latter usually require some getting used to.

2.8.1 rdply()

Necessary arguments for `rdply()` are the number of repetitions (`.n=`), and the function or expression to be repeated (`.expr=`). In the example below, the expression will be a user-defined function, but standard functions can also be used. There is an optional progress bar that indicates the progression of the repetitions, but this is not discussed until Chapter 7.

As an illustration of `rdply()`, suppose for each replication, $N = 10$ scores from a normal distribution are generated, and the mean is computed. This simulates the process of drawing a random sample from a known population and computing the sample mean each time. Such a simulation is common in introductory statistics courses to illustrate the sampling distribution of the mean.

The `rnorm()` function is used for generating data from a normal distribution. The arguments are the sample size (`n=`), the population mean (`mean=`), and the population SD (`sd=`). For this example, a population with $\mu = 100$ and $\sigma = 15$ is arbitrarily selected. For each sample, the mean is computed using `mean()`.

The function `mymean2` is defined to carry out the operations for each repetition. The function is executed twice below to illustrate that a different random sample is selected for each repetition.

```
> mymean2 <- function(){
+                      mysamp <- rnorm(n = 10, mean = 100, sd = 15)
+                      mean(mysamp)
+                     }
```

```
> mymean2()
```

```
[1] 90.63217
```

```
> mymean2()
```

```
[1] 108.9655
```

Assume the goal is to repeat the `mymean2()` function 15 times. The goal could be accomplished by executing 15 different lines of the same syntax. However, this is time-consuming and the resulting means are not stored in any convenient manner.

A better option is to use `rdply()`, which stores the output in a data frame object. The output object will record the repetition number and the resulting mean. The syntax `require(ggplot2)` is used to load the `ggplot2` and `plyr` packages. The `set.seed()` function with the arbitrary value 111 is included, so that the results can be replicated by the reader. In practice, the `set.seed()` function is not used unless the results need to be replicated.

```
> require(ggplot2)
> set.seed(111)
> myresult ← rdply(.n = 15, .expr = mymean2())
> colnames(myresult) ← c("rep", "mean")
> myresult
```

```
   rep      mean
1    1  99.42380
2    2 102.14509
3    3  89.36441
4    4 104.51982
5    5 107.68322
6    6 107.04305
7    7 103.34770
8    8  95.59829
9    9  98.98696
10  10  99.15023
11  11  99.65499
12  12 107.91997
13  13  96.24939
14  14 106.01613
15  15  91.50929
```

The output shows the replication number in the `rep` column, and the sample mean in the `mean` column. Consistent with statistical theory (see e.g., Howell, 2010, chap.4), the means appear to vary about $\mu = 100$.

To gain additional insight, consider a simulation using 15000 samples rather than 15. For each sample, the mean is computed and stored. After this is performed, the `ggplot()` function from the `ggplot2` package is used to create a density graph of the means.

In the next chapter the details of `ggplot()` are thoroughly discussed. The minimal elements required to create a density graph are shown below. The `data=` argument specifies the data frame, the `aes()` component defines the horizontal axis (x-axis) variable for the graph, and `geom_density()` draws the density curve. The `ggplot()` syntax is enclosed in parentheses forcing the automatic printing of the graph. When the syntax is executed, the graph will appear in the R graph window. The menu options in this window allow for the copying and saving of the graph.

```
> myresult ← rdply(.n = 15000, .expr = mymean2())
> colnames(myresult) ← c("rep", "mean")
> (ggplot(data = myresult, aes(x = mean)) + geom_density())
```

The density graph is shown in Figure 2.1. The distribution is approximately centered about $\mu = 100$, and the shape is consistent with a normal distribution.

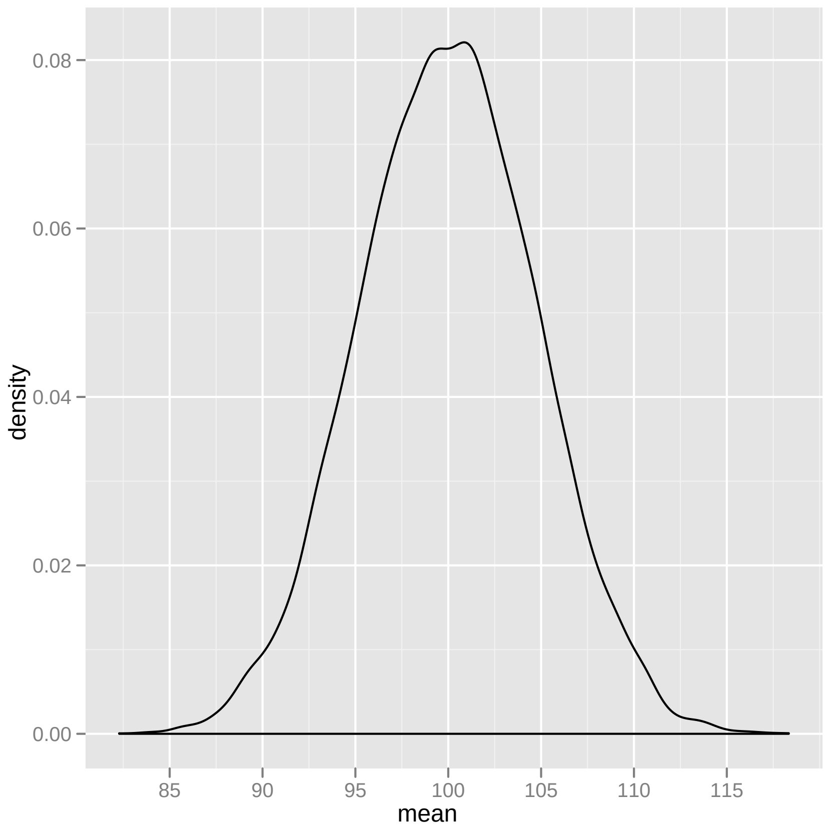
2.8.2 `for()` Loop

For certain bootstrap applications discussed later in this book, it is necessary to have greater control than that afforded by `rdply()`. In such cases, an alternative method of executing repetitive operations is used, the `for()` loop.

The `for()` loop has the basic form,

`for(replications){operations}`.

Figure 2.1: Density graph of means based on simulation.



The number and nature of the replications are defined in the parentheses, and the operations to be performed at each repetition are defined in the curly brackets. For this example, the operations are similar to the user-defined function considered above.

What makes the `for()` loop more difficult than `rdply()` for the novice programmer, is that the details of the repetitions must be specified with some rather esoteric conventions. An example is `for(i in 1:15)`, which will increment the `i` index by 1 starting at 1 and ending at 15 (i.e., $1, 2, 3, \ldots, 13, 14, 15$). This will provide 15 repetitions for the operations defined in the curly brackets.

A second difficulty is that, in contrast to `rdply()`, one must define a storage data frame prior to the `for()` loop, and assign the results of the operations for each repetition. The assignment is performed using the indexes discussed earlier in this chapter.

To fix ideas, consider the repetition of the sample mean simulation that was first performed with `rdply()`. The syntax below performs 15 replications. For each one, a sample of $N = 10$ is generated from a normal distribution, the sample mean is computed, and the repetition number and sample mean are stored in the data frame `myresult`. A twist here is that `myresult` is defined prior to the `for()` loop with its elements set to `NA`. The missing values are replaced with the repetition number and the sample mean within the loop.

```
> myresult ← as.data.frame(matrix(NA, ncol = 2, nrow = 15))   # Define storage data frame.
> for(i in 1:15){                                             # Repetition details.
+                 set.seed(i)                                 # Allows replication.
+                 mysamp ← rnorm(n = 10, mean = 100, sd = 15) # Generate sample data.
+                 myresult[i,1] ← i                           # Store repetition number.
+                 myresult[i,2] ← mean(mysamp)                # Store sample mean.
+                }
> colnames(myresult) ← c("rep", "mean")
> myresult
```

```
   rep      mean
1    1 101.98304
2    2 103.16727
3    3  98.99296
4    4 108.49794
5    5  98.81723
6    6 101.58037
7    7 101.55963
8    8  93.26021
9    9  96.71472
10  10  92.64015
11  11  96.54906
12  12  92.99179
13  13 109.00081
14  14 111.31309
15  15 102.67989
```

The output is the same form as that of `rdply()`, with the repetition number in the first column and the sample mean in the second column. The values of the means are different than in the `rdply()` example because `set.seed()` is used within the loop.

2.9 Linear Regression

Having discussed some basics of R programming, a more advanced example is included as a harbinger of things to come. Without much detail, a short traditional regression analysis is presented using made-up data. A more thorough treatment of traditional regression is given in Chapter 5.

Consider the two-predictor regression equation,

$$y = \beta_0 + \beta_1(x_1) + \beta_2(x_2) + \varepsilon, \tag{2.9.1}$$

where y is the response variable, x_1 and x_2 are the predictors, β_0 is the intercept, β_1 and β_2 are the regression coefficients (regression weights), and ε is a random error term.

The regression coefficients are estimated using the `lm()` function. Traditional regression is a special case of the *linear model* (LM), which explains the `lm()` abbreviation. The `lm()` function requires the names of the variables, and replaces the equal sign in the Equation (2.9.1) with a tilde ($\sim$).

In the syntax below, a sample size of $N = 20$ is generated for a response variable and two predictors. The `set.seed()` function is used so that the same random data can be generated by the reader in order to reproduce the example. The `rnorm()` function is used to generate the variables according to a normal distribution with $\mu = 100$ and $\sigma = 15$.

After generating the data, the parameters are estimated with `lm()`, and the output is saved as the object `lm.out`. The `summary()` function is used on the output object to display pertinent information.

```
> set.seed(123)                                 # Allows reproduction of results.
> y <-  rnorm(n = 20, mean = 100, sd = 15)      # Generate variables.
> x1 <- rnorm(n = 20, mean = 100, sd = 15)
> x2 <- rnorm(n = 20, mean = 100, sd = 15)
> lm.out <- lm(y ~ x1 + x2)                     # Estimate model and save output.
> summary(lm.out)                               # Display output.
```

```
Call:
lm(formula = y ~ x1 + x2)

Residuals:
    Min      1Q  Median      3Q     Max
-15.887 -10.287  -1.550   9.395  20.044

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept) 120.72000   28.30410   4.265 0.000523 ***
x1            0.05128    0.19302   0.266 0.793673
x2           -0.23315    0.21167  -1.101 0.286035
```

```
13 ---
14 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
15
16 Residual standard error: 12.58 on 17 degrees of freedom
17 Multiple $R^2$: 0.06826,    Adjusted $R^2$: -0.04136
18 F-statistic: 0.6227 on 2 and 17 DF,  p-value: 0.5483
```

Lines 1-2 show the syntax that was used. Lines 4-6 show descriptive statistics for the residuals. Lines 8-14 show information about the intercept and regression coefficients; the estimated values, SEs, t-ratios, and p-values. This information is provided for the intercept on line 10, for x_1 on line 11, and for x_2 on line 12. Line 14 provides a key for interpreting the symbols attached to the p-values (e.g., `*`). Lines 16-18 show information about omnibus fit; the residual standard error (line 16), R^2 and adjusted R^2 (line 17), and the omnibus F-test (line 18).

Additional details of the `lm()` output are provided in Chapter 5. For the moment, it is stressed that the output of a statistical function can be saved as an object, and additional functions can act on the saved object. The advantages of these features are illustrated throughout the book.

2.10 Getting Help

There are a number of methods for obtaining help with R. If the name of the function for which one wants help is known, then typing a question mark (`?`) and the name of the function at the prompt will display its help page. For example, `?mean` will display the help page for the `mean()` function. The help page shows the syntax conventions for the function that include optional arguments. Most helpful to new users are the examples that appear at the bottom of the help page.

For packages of known name, the `help()` function can be used. Typing, for example, `help(package=lme4)` at the prompt will display the functions available in the `lme4` package, supporting material, and technical specifications, if applicable.

The above two help functions require a user to know the name of the function or package for which they are requesting help. There is a potential for frustration, as one might feel there is an expectation to know what one does not know! To ease frustration, a topical search is recommended, as this only requires keywords for the topic of interest, for example, "standard deviation". A topical search is conducted with the `RSiteSearch()` function in the package of the same name. The function uses the search engine at `http://search.r-project.org` (the search engine can be used directly with a Web browser). A quoted topic must be supplied for the function, and the computer must be connected to the Internet.

Assume one wants help on the general topic of random effects. After submitting `require(RSiteSearch)`, the syntax below will open a web browser with listings that contain the quoted string,

```
RSiteSearch("random effects").
```

Without any options, the `RSiteSearch()` function displays help pages that contain the quoted string. Articles in the *R-help mailing list* can also be searched by adding the option, `restrict="Rhelp02a"`. R manuals can be searched by using the option `restrict="docs"`. The results can also be sorted, see `?RSiteSearch` for more details.

2.11 Summary of Functions

A summary of the functions discussed in this chapter appears in Table 2.2. The functions are loosely grouped based on general purpose categories. In addition to a brief description of each function, one or two key arguments are provided. The arguments may be optional depending on the function. For a more extensive list of options, the help page for the function should be consulted. In the remaining chapters, additional functions and arguments are presented as the need arises.

Table 2.2: List of R functions.

Function	Description	Arguments
Administrative		
`q()`	Quit	
`ls()`, `objects()`	List active objects	`pattern=`
`rm()`	Remove active objects	`list=ls()`
`help()`	Help for topic or package	`package=`
`?`	Help for a topic	
`RSiteSearch()`	Topical Internet search	`restrict=`
`install.packages()`	Install add-on packages	`repos=`
`library()`, `require()`	Load an installed package	
Statistical		
`mean()`	Mean	`na.rm=`
`sd()`	Standard deviation	`na.rm=`
`summary()`	Basic statistics	`digits=`
`min()`, `max()`	Minimum and maximum	`na.rm=`
`sum()`	Sum	`na.rm=`
`length()`	Length of a vector	
`set.seed()`	Set random number seed	
`rnorm()`	Generate normal data	`n=`, `mean=`, `sd=`
`lm()`	Linear regression	`y~x`, `data=`
Data Structure		
`c()`	Concatenation	
`matrix()`	Create matrix	`nrow=`, `ncol=`
`data.frame()`	Create data frame	`var.name=`
`list()`	Create a list	`vector.name=`
`colnames()`	Access data frame variable names	
`rownames()`	Access data frame row names	
`factor()`	Create factor variable	`labels=`
`str()`	Structure	`digits.d=`
`order()`	Sorting	`decreasing=`
`read.table()`	Read external text file	`file=`
Printing		
`print()`	Explicit print	`quote=`
Accessing Variables		
`attach()`, `detach()`	Use data frame variable names directly	
`with()`	Use data frame variable names directly	
Saving and Retrieving		
`save()`	Save objects in Rdata file	`file=`
`load()`	Load Rdata file	`file=`
Programming		
`function()`	User-defined functions	`x`
`rdply()`	Repetitive execution	`.n=`, `.expr=`
`for()`	Programming loop	