1

INTRODUCTION

Why are statistical methods important? One reason is that they play a fundamental role in a wide range of disciplines including physics, chemistry, astronomy, manufacturing, agriculture, communications, pharmaceuticals, medicine, biology, kinesiology, sports, sociology, political science, linguistics, business, economics, education, and psychology. Basic statistical techniques impact your life.

At its simplest level, statistics involves the description and summary of events. How many home runs did Babe Ruth hit? What is the average rainfall in Seattle? But from a scientific point of view, it has come to mean much more. Broadly defined, it is the science, technology, and art of extracting information from observational data, with an emphasis on solving real-world problems. As Stigler (1986, p. 1) has so eloquently put it:

> Modern statistics provides a quantitative technology for empirical science; it is a logic and methodology for the measurement of uncertainty and for examination of the consequences of that uncertainty in the planning and interpretation of experimentation and observation.

To help elucidate the types of problems addressed in this book, consider an experiment aimed at investigating the effects of ozone on weight gain in rats (Doksum and Sievers, 1976). The experimental group consisted of 22 seventy-day-old rats kept in an ozone environment for 7 days. A control group of 23 rats, of the same age, was kept in an ozone-free environment. The results of this experiment are shown in Table 1.1.

How should these two groups be compared? A natural reaction is to compute the average weight gain for both groups. The averages turn out to be 11 for the ozone group and 22.4 for the control group. The average is higher for the control group

Understanding and Applying Basic Statistical Methods Using R, First Edition. Rand R. Wilcox.

www.wiley.com/go/Wilcox/Statistical_Methods_R

TABLE 1.1 Weight Gain of Rats in Ozone Experiment

Control	41.0	38.4	24.4	25.9	21.9	18.3	13.1	27.3	28.5	−16.9
Ozone	10.1	6.1	20.4	7.3	14.3	15.5	−9.9	6.8	28.2	17.9
Control	26.0	17.4	21.8	15.4	27.4	19.2	22.4	17.7	26.0	29.4
Ozone	−9.0	−12.9	14.0	6.6	12.1	15.7	39.9	−15.9	54.6	−14.7
Control	21.4	26.6	22.7							
Ozone	44.1	−9.0								

suggesting that for the typical rat, weight gain will be less in an ozone environment. However, serious concerns come to mind upon a moment's reflection. Only 22 rats were kept in the ozone environment, and only 23 rats were in the control group. Suppose 100 rats had been used, or 1,000, or even a million. Is it reasonable to conclude that the ozone group would still have a smaller average than the control group? What about using the average to reflect the weight gain for the typical rat? Are there other methods for summarizing data that might have practical value when characterizing the differences between the groups? A goal of this book is to introduce the basic tools for answering these questions.

Most of the basic statistical methods currently taught and used were developed prior to the year 1960 and are based on strategies developed about 200 years ago. Of particular importance was the work of Pierre-Simon Laplace (1749–1827) and Carl Friedrich Gauss (1777–1855). Approximately a century ago, major advances began to appear, which dominate how researchers analyze data today. Especially important was the work of Karl Pearson (1857–1936) Jerzy Neyman (1894–1981), Egon Pearson (1895–1980), and Sir Ronald Fisher (1890–1962). For various reasons summarized in subsequent chapters, it was once thought that these methods generally perform well in terms of extracting accurate information from data. But in recent years, it has become evident that this is not always the case. Indeed, three major insights have revealed conditions where methods routinely used today can be highly unsatisfactory.

The good news is that many new and improved methods have been developed that are aimed at dealing with known problems associated with the more commonly used techniques. In practical terms, modern technology offers the opportunity to get a deeper and more accurate understanding of data. So, a major goal of this book is to introduce basic methods in a manner that builds a conceptual foundation for understanding when commonly used techniques perform in a satisfactory manner and when this is not the case. Another goal is to provide some understanding of when and why more modern methods have practical value.

This book does *not* describe the mathematical underpinnings of routinely used statistical techniques, but rather the concepts and principles that are used. Generally, the essence of statistical reasoning can be understood with little training in mathematics beyond basic high school algebra. However, there are several key components underlying the basic strategies to be described, the result being that it is easy to lose track of where we are going when the individual components are being explained. Consequently, it might help to provide a brief overview of what is covered in this book.

1.1 SAMPLES VERSUS POPULATIONS

A key aspect of most statistical methods is the distinction between a sample of participants or objects and a population of participants or objects. A *population* of participants or objects consists of all those participants or objects that are relevant in a particular study. In the weight-gain experiment with rats, there are millions of rats that could be used if sufficient resources were available. To be concrete, suppose there are a billion rats and the goal is to determine the average weight gain if all 1 billion were kept in an ozone environment. Then, these 1 billion rats compose the population of rats we wish to study. The average gain for these rats is called the *population mean*. In a similar manner, there is an average weight gain for all 1 billion rats that might be raised in an ozone-free environment instead. This is the *population mean* for rats raised in an ozone-free environment. The obvious problem is that it is impractical to measure all 1 billion rats. In the experiment, only 22 rats were kept in an ozone environment. These 22 rats are an example of a *sample*.

Definition. A *sample* is any subset of the population of individuals or things under study.

Example

Imagine that a new method for treating depression is tried on 20 individuals. Further imagine that after treatment with the new method, depressive symptoms are measured and the average is found to be 16. So, we have information about the 20 individuals in the study, but of particular importance is knowing the average that would result if all individuals suffering from depression were treated with the new method. The population corresponds to all individuals suffering from depression. The sample consists of the 20 individuals who were treated with the new method. A basic issue is the uncertainty of how well the average based on the 20 individuals in the study reflects the average if all depressed individuals were to receive the new treatment.

Example

Shortly after the Norman Conquest, around the year 1100, there was already a need for methods that indicate how well a sample reflects a population of objects. The population of objects in this case consisted of coins produced on any given day. It was desired that the weight of each coin be close to some specified amount. As a check on the manufacturing process, a selection of each day's coins was reserved in a box ("the Pyx") for inspection. In modern terminology, the coins selected for inspection are an example of a sample, and the goal is to generalize to the population of coins, which in this case is all the coins produced on that day.

Three Fundamental Components of Statistics Statistical techniques consist of a wide range of goals, techniques, and strategies. Three fundamental components worth stressing are given as follows:

1. *Design*. Roughly, this refers to a procedure for planning experiments so that data yield valid and objective conclusions. Well-chosen experimental designs

maximize the amount of information that can be obtained for a given amount of experimental effort.

2. *Description*. This refers to numerical and graphical methods for summarizing data.
3. *Inference*. This refers to making predictions or generalizations about a population of individuals or things based on a sample of observations.

Design is a vast subject, and only the most basic issues are discussed here. The immediate goal is to describe some fundamental reasons why design is important. As a simple illustration, imagine you are interested in factors that affect health. In North America, where fat accounts for a third of the calories consumed, the death rate from heart disease is 20 times higher than in rural China where the typical diet is closer to 10% fat. What are we to make of this? Should we eliminate as much fat from our diet as possible? Are all fats bad? Could it be that some are beneficial? This purely descriptive study does not address these issues in an adequate manner. This is not to say that descriptive studies have no merit, only that resolving important issues can be difficult or impossible without good experimental design. For example, heart disease is relatively rare in Mediterranean countries where fat intake can approach 40% of calories. One distinguishing feature between the American diet and the Mediterranean diet is the type of fat consumed. So, one possibility is that the amount of fat in a diet, without regard to the type of fat, might be a poor gauge of nutritional quality. Note, however, that in the observational study just described, nothing has been done to control other factors that might influence heart disease. Sorting out what does and does not contribute to heart disease requires good experimental design.

The ozone experiment provides a simple example where design considerations are important. It is evident that the age of young rats is related to their weight. So, the experiment was designed to control for age by using rats that have the same age and then manipulate the *single* factor that is of interest, namely, the amount of ozone in the air.

Description refers to ways of summarizing data that provide useful information about the phenomenon under study. It includes methods for describing both the sample available to us and the entire population of participants if only they could be measured. The average is one of the most common ways of summarizing data. As previously noted, the average for all the participants in a population is called the *population mean*; it is typically represented by the Greek letter mu, μ. The average based on a sample of participants is called a *sample mean*. The hope is that the sample mean provides a good reflection of the population mean. Inferential methods described in subsequent chapters are designed to determine the extent this goal is achieved.

1.2 COMMENTS ON SOFTWARE

As is probably evident, a key component to getting the most accurate and useful information from data is software. There are now several popular software packages for analyzing data. Perhaps the most important thing to keep in mind is that the choice of software can be crucial, particularly when the goal is to apply new and improved

methods developed during the last half century. Presumably, no software package is perfect, based on all of the criteria that might be used to judge them, but the following comments about some of the choices might help.

Best Software The software R is arguably the best software package for applying all of the methods covered in this book. Moreover, it is free and available at http://cran.R-project.org. Many modern methods developed in recent years, as well as all classic techniques, can be applied. One feature that makes R highly valuable from a research perspective is that a group of statisticians does an excellent job of constantly adding and updating routines aimed at applying modern techniques. A wide range of modern methods can be applied using the basic package and many specialized methods are available, some of which are described in this book.

Very Good Software SAS is another software package that provides power and excellent flexibility. Many modern methods can be applied, but a large number of the most recently developed techniques are not yet available via SAS. SAS code could be written by anyone reasonably familiar with SAS, and the company is fairly diligent about upgrading the routines in their package, but this has not been done yet for some of the methods covered in this book.

Good Software Minitab is fairly simple to use and provides a reasonable degree of flexibility when analyzing data. All of the standard methods developed prior to the year 1960 are readily available. Many modern methods could be run in Minitab, but doing so is not straightforward. Similarly to SAS, special Minitab code is needed, and writing this code would take some effort. Moreover, certain modern methods that are readily applied with R cannot be easily applied when using Minitab even if an investigator is willing to write the appropriate code.

Unsatisfactory Software SPSS is certainly one of the more popular and frequently used software packages. But in terms of providing access to the many new and improved methods for comparing groups and studying associations, which have appeared during the last half century, it must be given a poor rating. An additional concern is its lack of flexibility compared to R. It is a relatively simple matter for statisticians to create specialized R code that provides non-statisticians with easy access to modern methods. Some modern methods can be applied with SPSS, but often this task is difficult or virtually impossible.

The software EXCEL is relatively easy to use, it provides some flexibility, but generally modern methods are not readily applied. McCullough and Wilson (2005) conclude that this software package is not maintained in an adequate manner. (For a more detailed description of some problems with this software, see Heiser, 2006.) Even if EXCEL functions were available for all modern methods that might be used, features noted by McCullough and Wilson suggest that EXCEL should not be used. Some improvements were made in Microsoft Excel 2010, but serious concerns remain (Mélard, 2014).

1.3 R BASICS

R is a vast and powerful software package. Even a book completely dedicated to R cannot cover all of the methods and features that are available. The immediate goal is

to describe the basic features that are needed in this book. Many additional features are introduced in subsequent chapters when they are required. Another resource for learning the basics of R is Swirl, which can be installed as described at http://swirlstats.com/. There are many books on R as a Google Search will reveal. A free manual is available at http://cran.r-project.org/. On the left side of this web page near the bottom under documentation, you will see a manual. Click on this link to gain access to the manual. Another free manual is available at http://cran.r-project.org/doc/contrib/Verzani-SimpleR.pdf. (Or, google Verzani's simpleR.)

R can be downloaded from http://cran.r-project.org/. At the top of the web page, you will see three links:

- Download R for Linux
- Download R for (Mac) OS X
- Download R for Windows

Simply click on the link matching the computer you are using to download R. Once you start R, you will see this prompt:

```
>.
```

This means that R is waiting for a command. (You do not type > The prompt is used to indicate where you type commands.) To quit R, use the command

```
> q().
```

That is, type `q()` and hit Enter.

1.3.1 Entering Data

To begin with the simplest case, imagine you want to store the value 5 in an R variable called `blob`. This can be done with the command

```
blob=5.
```

Typing `blob` and hitting Enter will produce the value 5 on the computer screen.

An important feature of R is that a collection of values can be stored in a single R variable. One way of doing this is with the R command `c`, which stands for combine. For example, the values 2, 4, 6, 8, 12 can be stored in the R variable `blob` with the command

```
blob=c(2,4,6,8,12).
```

The first value, 2, is stored in `blob[1]`, the second value is stored in `blob[2]`, and so on. To determine how many values are stored in an R variable, use the R command `length`. In the example

```
length(blob)
```

would return the value 5.

R has various ways of reading data stored in a file. The R commands `scan` and `read.table` are the two basic ways of accomplishing this goal. The simplest version of the `scan` command assumes that a string of values is to be read. By default, R

assumes that values are separated by one or more spaces. Missing values are assumed to be recorded as NA for "not available."

For example, imagine that a file called *ice.dat* contains

6 3 12 8 9.

Then, the command

```
ice=scan(file="ice.dat")
```

will read these values from the file and store them in the R variable `ice`. However, this assumes that the data are stored in the directory where R expects to find the file. Just where R expects to find the file depends on the hardware and how R is being accessed. (It might assume that data are stored in the main directory or in Documents, but other possibilities exist.) A simple way of dealing with this issue is to use the R command `file.choose` in conjunction with the scan command. That is, use the command

```
ice=scan(file.choose()).
```

This will open a window that lists the files on your computer. Simply click on the appropriate file.

If the data are not separated by one or more spaces, but rather some particular character, the argument `sep` tells R how the values are separated. For example, suppose the file contains

6, 3, 12, 8, 9.

That is, the values in the file are separated by a comma rather than a space. Now use the command

```
ice=scan(file.choose(),sep=",")
```

In other words, the argument `sep` tells R the character used to separate the values. If the values were separated by &, use the command

```
ice=scan(file.choose(),sep="&").
```

When you quit R with the command `q()`, R will ask whether you want to save your results. If you answer yes, R variables containing data will remain in R until they are removed. So, in this last example, if you quit R and then restart R, typing `ice` will again return the values 6, 3, 12, 8, 9. To remove data, use the `rm` command. For example,

```
rm(ice)
```

would remove the R variable `ice`.

R variables are case-sensitive. So, for example, the command

```
Ice=5
```

would store the value 5 in `Ice`, but the R variable `ice` would still contain the values listed previously, unless of course they had been removed.

The R command `read.table` is another commonly used method for reading data into R. It is designed to read columns of data where each column contains the values of some variable. It has the general form

```
read.table(file, header = FALSE, sep = "",na.strings =
"NA",skip=0)
```

where the argument file indicates the name of the file where the data are stored and the other arguments are explained momentarily. Notice that the first argument, `file`, does not contain an equal sign. This means that a value for this argument must be specified. With an equal sign, the argument defaults to the value shown. So, for `read.table`, `header` is an optional argument that will be taken to be `header=FALSE` if it is not used.

Example

Suppose a file called quake.dat contains three measures related to earthquakes:

```
magnitude length  duration
7.8       360     130
7.7       400     110
7.5        75      27
7.3        70      24
7.0        40       7
6.9        50      15
6.7        16       8
6.7        14       7
6.6        23      15
6.5        25       6
6.4        30      13
6.4        15       5
6.1        15       5
5.9        20       4
5.9         6       3
5.8         5       2
```

Then, the R command

```
quake=read.table('quake.dat',skip=1)
```

will read the data into the R variable `quake`, where the argument `skip=1` indicates that the first line of the data file is to be ignored, which in this case contains the three labels: magnitude, length, and duration. As with the `scan` command, you can point to the file to be read using the `file.choose` command. That is, use the command

```
quake=read.table(file.choose(),skip=1).
```

Typing `quake` and hitting Enter produce

```
   V1    V2    V3
1 7.8   360   130
2 7.7   400   110
```

```
 3 7.5   75    27
 4 7.3   70    24
 5 7.0   40     7
 6 6.9   50    15
 7 6.7   16     8
 8 6.7   14     7
 9 6.6   23    15
10 6.5   25     6
11 6.4   30    13
12 6.4   15     5
13 6.1   15     5
14 5.9   20     4
15 5.9    6     3
16 5.8    5     2
```

So, the columns of the data are given the names V1, V2, and V3, as indicated. Typing the R command

```
quake$V1
```

and hitting Enter would return the first column of data, namely the magnitude of the earthquakes. The command

```
quake[,1]
```

would accomplish the same goal. In a similar manner, `quake[,2]` contains the data in the second column and `quake[,3]` contains the data in the third column. In contrast, `quake[1,]` contains the data in the first row. So, typing the command

```
quake[1,]
```

and hitting Enter would return

```
7.8 360 130.
```

The command `quake[1,1]` would return the value 7.8, the value stored in the first row and the first column, and `quake[1,2]` would return 360.

Now consider the command

```
quake=read.table("quake.dat",header=TRUE).
```

The argument `header=TRUE` tells R that the first line in the file contains labels for the columns of data, which for the earthquake data are magnitude, length, and duration. Now the command

```
quake$magnitude
```

would print the first column of data on the computer screen. The R command

```
labels
```

returns the labels associated with an R variable. For the situation at hand,

```
labels(quake)
```

would return magnitude, length, and duration. The command

```
head(dat)
```

returns the first six lines of the data stored in `dat` and

```
tail(dat)
```

returns the final six lines. The R command

```
str(dat)
```

returns information about the number of observations, the number of variables, and it lists some of the values stored in `dat`. (It also indicates the storage mode of `dat`; storage modes are explained in Section 1.3.3.)

As was the case with the `scan` command, `read.table` assumes that values stored in a data file are separated by one or more spaces. But suppose the values are separated by some symbol, say &. Again, this can be handled via the argument `sep`. Now the command would resemble this:

```
quake=read.table("quake.dat",sep="&",header=TRUE)
```

Another common issue is reading data into R that are stored in some other software package such as SPSS. The seemingly easiest way of dealing with this is to first store the data in a .csv file, where csv stands for Comma Separated Values. (SPSS, EXCEL, and other popular software have a command to accomplish this goal.) Once this is done, the data can be read into R using the `read.csv` command, which is used in the same way as the `read.table` command.

Now imagine that some of the values in the quake file are missing and that missing values are indicated by M rather than NA. The argument `na.strings` can be used to indicate that missing values are stored as M. That is, now the data would be read using the R command

```
quake=read.table(quake.dat.txt,na.strings="M",header=TRUE).
```

1.3.2 Arithmetic Operations

In the simplest case, arithmetic operations can be performed on numbers using the operators +, −, * (multiplication), / (division), and ^ (exponentiation). For example, to compute 1 plus 5 squared, use the command

```
1+5^2
```

which returns

```
[1] 26.
```

To store the answer in an R variable, say `ans`, use the command

```
ans=1+5^2.
```

If two or more values are stored in an R variable, arithmetic operations applied to the variable name will be performed on all the values. For example, if the values 2, 5, 8, 12, and 25 are stored in the R variable `vdat`, then the command

```
vinv=1/vdat
```

will compute 1/2, 1/5, 1/8, 1/12, and 1/25, and store the results in the R variable `vinv`. The R command

```
2*vdat
```

multiplies every value in `vdat` by 2. For the situation at hand, it returns 4, 10, 16, 24, and 50. The command

```
vdat-2
```

returns 0, 3, 6, 10, and 23. That is, 2 is subtracted from every element in `vdat`.

Most R commands consist of a name of some function followed by one or more arguments enclosed in parentheses. There are hundreds of functions that come with R. Some of the more basic functions are listed as follows:

Function	Description
abs	Absolute value
exp	Exponential
log	Natural logarithm
sqrt	Square root
cor	Correlation (explained in Chapter 8)
mean	Arithmetic mean (with a trimming option)
median	Median (explained in Chapter 2)
min	Smallest value
max	Largest value
range	Determines the minimum and maximum values
sd	Standard deviation (explained in Chapter 2)
sum	Arithmetic sum
var	Variance (explained in Chapter 2) and covariance (explained in Chapter 8)
length	Indicates how many values are stored in an R variable.

Example

If the values 2, 7, 9, and 14 are stored in the R variable `x`, the command

```
min(x)
```

returns 2, the smallest of the four values stored in x. The average of the numbers is computed with the command

```
mean(x),
```

which returns the value 8. The command `range(x)` returns the largest and smallest values stored in `x`, which are 2 and 14, respectively. The command

```
v=range(x)
```

would result in the smallest value being stored in `v[1]` and the the largest value stored in `v[2]`. The command

```
sum(x)
```

returns the value $2 + 7 + 9 + 14 = 32$. The command

```
sum(x)/length(x)
```

is another way to compute the average.

1.3.3 Storage Types and Modes

R has several ways of storing data. The four types that are important in this book are vectors, matrices, data frames, and list mode.

Vectors are the most basic way of storing data. A vector is just a collection of values stored in some R variable. When data are read into R using the `scan` command, the data are stored in a vector. The R command

```
blob=c(2,4,6,8,12),
```

previously described, creates a vector containing the values 2, 4, 6, 8, and 12.

A matrix is a rectangular array of values having n rows and J columns. Imagine, for example, that the performance of 10 athletes is measured on three different occasions. Then, a convenient way of storing the data is in a matrix with $n = 10$ rows and $J = 3$ columns. So, the first row contains the three measures for the first individual, the second row contains the three measures for the second individual, and so on.

As another example, imagine that for each of 20 individuals you measure blood glucose levels, anxiety, height, and weight. Then, a convenient way of storing the data is in a matrix with $n = 20$ rows and $J = 4$ columns, where the first column contains blood glucose levels, the second column contains a measure of anxiety, and so forth. A vector can be thought of as a matrix with a single column, but R makes a distinction between a vector and a matrix. If, for example, `x` is a matrix with 10 rows and 2 columns, the R command

```
is.vector(x)
```

returns FALSE, meaning that the R variable `x` is not a vector. But the R command

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

returns TRUE, meaning that column 1 of the matrix `x` is a vector. More generally, any single column of a matrix is a vector. The same is true for any row of `x`. For example,

```
is.vector(x[2,])
```

would return TRUE, meaning that the second row of x is a vector. In contrast,

```
is.matrix(x)
```

returns TRUE and the command

```
is.matrix(x[,1])
```

returns FALSE.

There are several ways a matrix can be created in R. For example, imagine that you already have an R variable BPS containing measures of systolic blood pressure for 30 individuals and for the same 30 individuals, the R variable BPD contains their diastolic blood pressure. The command

```
BP=cbind(BPS,BPD)
```

would result in a matrix with 30 rows and 2 columns. The command `cbind(BPS, BPD)` says to bind together BPS and BPD as columns. The R command `rbind` binds together rows.

To determine how many rows a matrix has, use the R command `nrow`. The command `ncol` returns the number of columns and `dim` returns both the number of rows and columns. Here, `ncol(BP)` would return the value 2.

You can apply arithmetic operations to specific rows or columns of a matrix. For example, to compute the average of all values in column 1 of the matrix `m`, use the command

```
mean(m[,1]).
```

The command

```
mean(m[2,])
```

computes the average of all values in row 2. In contrast, the command

```
mean(m)
```

would average all of the values in `m`. If you have several columns of data and want to compute the average for each column, there is a quick way of doing this via the `apply` command. The command

```
apply(m,2,mean)
```

computes the average for each column. The command

```
apply(m,1,mean)
```

computes the average for each row.

Another way to create a matrix is with the R command `matrix`.

Example

The command

```
matrix(c(1,4,8,3,7,2),ncol=2)
```

returns

```
     [,1] [,2]
[1,]    1    3
[2,]    4    7
[3,]    8    2
```

The argument `ncol` tells R how many columns the matrix is to have. Note that the first three numbers are stored in the first column and the other three are in column 2. In contrast, the command

```
matrix(c(1,4,8,3,7,2),ncol=2,byrow=TRUE)
```

returns

```
     [,1] [,2]
[1,]    1    4
[2,]    8    3
[3,]    7    2
```

Now the first two numbers are stored in the first row, the next two are stored in the second row, and the final two are stored in the third row. That is, it fills in the numbers by rows rather than columns.

There are several storage modes used by R. The four important ones in this book are the following: logical, numeric, character, and factor. As the name implies, numeric refers to numbers in contrast to characters such as "A." (Logical variables are discussed in Section 1.3.4.) For present purposes, factor variables can be viewed as a storage mode that indicates the group to which an individual belongs. Imagine, for example, a study dealing with hypertension where some individuals receive an experimental drug and others receive a placebo. Further imagine that the data are stored in the file BP_study containing two columns. The first column contains blood pressure measures and the second contains the letters E and P, where E indicates that an individual received the experimental drug and P indicates a placebo. So, the first few rows in the file might resemble this:

```
120  E
145  P
132  E
121  E
139  P
```

As previously explained, the R command

```
BP=read.table(file="BP_study")
```

can be used to read the data and store it in the R variable BP. When this is done, BP will be stored in what R calls a *data frame*. Similar to a matrix, a data frame consists of n rows and J columns. The advantage of a data frame over a matrix is that different columns of a data frame can contain different types of data. In contrast, a matrix cannot have a mix of data types. For example, it cannot contain both numeric and character data: it must be all numeric, or it can be all characters. A data frame is more flexible in the sense that some columns can have numeric data that are to be analyzed and other columns can be something other than numeric data such as a factor variable. When the R command read.table encounters a column of data in a file that is not numeric, it stores it as a factor. In the example, the second column would be read as a factor variable with two possible values: E and P.

When data are stored in a data frame, with some column having the factor mode, situations are encountered where it will be necessary to separate the data into R variables based on the values of the factor variable. In the last example, it might be necessary to separate the participants receiving the experimental drug from those who received the placebo. There are several ways to do this, but the details are postponed until they are needed. (Chapter 9 illustrates this process; see the discussion of the R commands split and fac2list.)

List mode is yet another convenient way of storing data. It will play an important role in Chapters 9–11. Imagine that three groups of participants are to be compared: those with little or no signs of depression, those with mild depression, and those with severe depression. Further suppose that there are 20, 30, and 10 individuals belonging to these groups, respectively, and that the data are stored in the R variables G1, G2, and G3. For various purposes, it can be convenient to store the data for all three groups in a single R variable. A matrix is not convenient because the number of individuals differs among the three groups. A more convenient way to combine the data into a single R variable is to use list mode. In this book, the most common way of storing data in list mode is via the R function fac2list, or the R function split, which are described and illustrated in Chapter 9. In case it helps, here is another way this can be done:

```
DEP=list()
DEP[[1]]=G1
DEP[[2]]=G2
DEP[[3]]=G3
```

The average of the data in the second group, for example, can be computed with the R command

```
mean(DEP[[2]]).
```

If the goal is to compute the average for all three groups, there is an easier way to do this via the R command lapply. The command

```
lapply(DEP,mean)
```

accomplishes this goal.

One more way of storing data in list mode is as follows. First create a variable having list mode. If you want the variable to be called `gdat`, use the command

```
gdat=list()
```

Then, the data for group 1 can be stored via the command

```
gdat[[1]]=c(36, 24, 82, 12, 90, 33, 14, 19)
```

the group 2 data would be stored via the command

```
gdat[[2]]=c(9, 17, 8, 22, 15)
```

and the group 3 data would be stored by using the command

```
gdat[[3]]=c(43, 56, 23, 10)
```

Typing the command `gdat` and hitting Enter return

```
[[1]]:
[1] 36 24 82 12 90 33 14 19

[[2]]:
[1]  9 17  8 22 15

[[3]]:
[1] 43 56 23 10
```

That is, `gdat` contains three vectors of numbers corresponding to the three groups under study.

Another way to store data in list mode is with a variation of the scan command. Suppose the data are stored in a file called *mydata.dat* and are arranged as follows:

```
36  9  43
24  17 56
82  8  23
12  22 10
90  15 NA
33  NA NA
14  NA NA
19  NA NA
```

Then, the command

```
gdat=scan("mydata.dat",list(g1=0,g2=0,g3=0))
```

will store the data in `gdat` in list mode. Typing `gdat` and hitting Enter return

```
$g1:
[1] 36 24 82 12 90 33 14 19
$g2:
[1] 9 17 8 22 15 NA NA NA
$g3:
[1] 43 56 23 10 NA NA NA NA
```

So, the data for group 1 are stored in `gdat$g1`; for group 2, they are in `gdat$g2`; and for group 3, they are in gdat$g3. An alternative way of accessing the data in group 1 is with `gdat[[1]]`. Note that when scan is used to store data in list mode, it assumes that the data for group 1 are stored in column 1, group 2 data are stored in column 2, and group 3 data are stored in column 3.

1.3.4 Identifying and Analyzing Special Cases

There are various ways R can be used to examine a subset of the data. The earthquake data, described at the end of Section 1.3.1, provide a glimpse of why this can be important and useful. Still assuming the data are stored in the R variable `quake`, the R command `mean(quake[,2])` indicates that the average of the length variable (the average of the data in column 2) is 72.75. Note that among the 16 observations, only 3 have a value greater than 72.75. Also notice that the two largest length values, 360 and 400, appear to be unusually large compared to the other values. Do these two values have an inordinate impact on the average? A way of answering this question is to eliminate these two values and averaging the remaining data. Noting that the two largest values are stored in rows 1 and 2, this can be accomplished by averaging the data in rows 3–16 of data frame `quake`. An R command that accomplishes this goal is

```
mean(quake[3:16,2]).
```

Now the average is only 28.86, suggesting that the two most extreme values have an inordinate impact on the average. In `quake[3:16,2]`, the command `3:16` indicates that rows 3–16 of `quake` are to be used. Another but more tedious way of averaging over rows 3–16 is as follows:

```
id=c(3,4,5,6,7,8,9,10,11,12,13,14,15,16)
mean(quake[id,2]).
```

In this particular case, it was a simple matter to identify and ignore the rows of data that seem to be having an inordinate impact on the average length. But often, some alternative method for selecting special rows of data is required and *logical variables* provide a convenient way of accomplishing this goal. Again using the quake data, the following R commands illustrate one way of doing this:

```
flag=quake[,2]<360
mean(quake[flag,2]).
```

The first command creates a logical variable that contains TRUE or FALSE, depending on whether the length values (the values in column 2) are less than 360. That is, the command `flag=quake[,2]<360` returns TRUE for any value in column 2 that is less than 360. Typing the R command `flag` and hitting Enter return

FALSE FALSE TRUE TRUE TRUE TRUE TRUE TRUE TRUE TRUE
TRUE TRUE TRUE TRUE TRUE TRUE

That is, `flag[1]=FALSE`, `flag[2]=FALSE`, `flag[3]=TRUE`, and so forth. The command

```
mean(quake[flag,2])
```

tells R to compute the mean using only those rows of `quake` for which `flag` is TRUE. Another way to accomplish the same goal is to use the R command

```
flag=which(quake[,2]<360).
```

Now `flag` indicates the row numbers for which the length is less than 360. That is, the values in `flag` are 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16. Yet another way is to eliminate or ignore the first two rows in `quake` when computing the average of the values in column 2. This can be done with the R command

```
mean(quake[c(-1,-2),2])
```

which again returns the value 28.86. That is, row numbers that are negative are ignored.

As just indicated, the command `quake[flag,2])` tells R to use only those rows of `quake` for which `flag` is TRUE. The command `quake[!flag,2])` tells R to use only those rows for which `flag` is FALSE. That is, the command ! means "not."

Logical variables provide one way of dealing with missing values. The R function `is.na` determines which values are missing, which can be used to eliminate them.

Example

Consider the R command

```
z=c(45, 23, NA, 19 , 12)
```

So, the third value, stored in `z[3]`, is missing. The R command

```
is.na(z)
```

returns FALSE FALSE TRUE FALSE FALSE, where FALSE means the corresponding value is not missing, and TRUE means that it is missing. Remembering that ! means not, the command

```
flag=!is.na(z)
```

indicates which values are not missing. So, for example, `flag[1]=TRUE`, `flag[3]=FALSE`, and `z[flag]` contains 45, 23, 19, and 12. An even simpler way of eliminating missing values stored in `z` is with the command

```
z=z[!is.na(z)].
```

TABLE 1.2 A Summary of Basic Commands for Accessing Data When Working with a Vector x

`x[4]`	Access the fourth element in x
`x[c(1,4,7)]`	Access the first, fourth, and seventh elements in x
`x[c(-1,-4,-7)]`	Access all elements in x excluding elements 1, 4, and 7
`x[x>5]`	Access all elements in x having a value greater than 5
`x[x>=5]`	Access all elements in x having a value greater than or equal to 5
`x[x>=5 | x<0]`	Access all elements in x having a value greater than 5 or less than 0
`x[abs(x)>12 & x<8]`	Access all elements in x that have an absolute value greater than 12 and a value less than 8.
`x[!is.na(x)]`	Access all elements in x that are not missing
`which(x==min(x))`	Indicates which elements of x contain the smallest value

Table 1.2 summarizes some basic commands for accessing data when working with a vector x. The last six methods in Table 1.2 make use of logical variables that can be very helpful when manipulating data.

Example

The following illustrates the last command in Table 1.2:

```
x=c(8,12,9,2,23,19)
which(x==min(x)).
```

The second command, `which(x==min(x))`, returns the value 4, meaning that x[4] contains the smallest value.

Example

For the situation in the last example,

```
z=min(x)
```

stores a single value in z, namely the smallest value in x, which is 2. In contrast, the command

```
z=x==min(x)
```

stores

FALSE FALSE FALSE TRUE FALSE FALSE

in the R variable z. The command

```
x[z]
```

returns only those values in x for which the corresponding value in z is TRUE, which in this case is the value 2.

Example

To illustrate the next to last command in Table 1.2, suppose the values 22, NA, 13, 6, 19, 23 are stored in the R variable x. Then, the R command mean attempts to compute the average, but it returns NA because the second observation is missing (not available). One way to remove any missing values and compute the average of the remaining data is with the command

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

Another way is to use the command

```
mean(x,na.rm=TRUE).
```

1.4 R PACKAGES

R has many built-in functions for applying the methods that are routinely taught and used. We will see, however, that these classic techniques can miss important features of data that are revealed when using more modern methods. An extremely important feature of R is that more modern methods are readily applied via freely available R packages that have been written by numerous statisticians. Literally, hundreds of new and improved techniques can now be used to gain a deeper and more accurate understanding of data. A few of these more modern methods are described in this book and will be seen to have considerable practical importance.

R packages are available from two sources. The first is located on the web at r-forge .r-project.org. The other is CRANS, located at cran.r-project.org. R packages available from CRANS can be installed with the R command install.packages. For example, the R command

```
install.packages("akima")
```

will install the R package akima, which is used when creating three-dimensional plots.

There is a particular R package that plays a central role in this book, and there are two ways it can be installed. The first and simplest method is to download the file Rallfun, which is stored at

http://dornsife.usc.edu/labs/rwilcox/software/.

The current version is labeled Rallfun-v28. Once the file is downloaded, use the R command

```
source("Rallfun-v28")
```

to execute the R commands stored in Rallfun-v28. This assumes that Rallfun-v28 is stored where R expects to find data. If Rallfun-v28 is not stored where R expects to find data, use the source command in conjunction with the file.choose command. That is, use the R command

```
source(file.choose()).
```

A previously explained, `file.choose` provides a way of finding where the file Rallfun-v28 is stored. By clicking on this file, the functions written for this book will be incorporated into your version of R. This file contains over 1,100 R functions for applying modern statistical methods.

A second way of gaining access is via the R package WRS (maintained by Felix Schönbrodt). Go to

https://github.com.

At the top of the web page, you will see Search GitHub. Type WRS and hit return. You can either download the Rallfun file, or you can install the WRS package using the R commands indicated on this web page. These R commands can also be located at

https://github.com/nicebread/WRS

Copy and paste the R commands into R. Then, use the R command

```
library(WRS)
```

to gain access to the functions.

A subset of the R functions in Rallfun is available in the R package WRS2 (created by Patrick Mair), which is stored on CRANS. A possible appeal of this package is that it contains help files and it is easily installed by using the R command

```
install.packages("WRS2").
```

The R command

```
library(WRS2)
```

provides access to the functions, and it lists the functions that are available. Currently, a negative feature is that WRS2 does not contain all of the functions described and illustrated in this book.

Information about built-in R functions, as well as functions in R packages downloaded from CRANS, can be obtained by typing the R command ? followed by the name of the function. For example, to get a brief summary of the R function `mean`, use the R command

```
?mean.
```

A window will appear that summarizes what the function does. For the functions in Rallfun, a different approach is required: type the name of the function and hit Enter. For example, there is a function called `yuen`, which is described in Chapter 9. The R command

```
yuen
```

lists the R code on your computer screen. At the top of the code, you will see

```
#
#  Perform Yuen's test for trimmed means on the data in x and y.
#  The default amount of trimming is 20%
#  Missing values (values stored as NA) are automatically removed.
#
#  A confidence interval for the trimmed mean of x minus the
#  the trimmed mean of y is computed and returned in yuen$ci.
#  The $p$-value is returned in yuen$p. value
#
```

These first few lines provide a quick summary of what the function does and how it is used. To see a list of the arguments used by the function, use the `args` command. For example,

```
args(yuen)
```

returns

```
function (x, y, tr = 0.2, alpha = 0.05).
```

So, the function expects to find data stored in two variables, labeled here as `x` and `y`. There are two optional arguments. The first is `tr`, which defaults to 0.2, and the second, `alpha`, which defaults to 0.05, both of which are explained in subsequent chapters.

Many of the R functions written for this book are based in part on other R packages available at CRANS. They include the following packages:

- akima
- MASS
- mgcv
- plotrix
- quantreg
- robust
- rrcov
- scatterplot3d
- stats

All of these packages can be installed with the `install.packages` command (assuming that you are connected to the web).

1.5 ACCESS TO DATA USED IN THIS BOOK

R has many built-in data sets. To see a list of the data sets that are available, type the R command

```
data().
```

Additional data sets used to illustrate the methods in this book can be downloaded from

http://dornsife.usc.edu/labs/rwilcox/datasets/.

1.6 ACCESSING MORE DETAILED ANSWERS TO THE EXERCISES

For each chapter in this book, brief answers to most of the exercises are located in Appendix A. More detailed answers for all of the exercises are located at http://dornsife.usc.edu/labs/rwilcox/books/ and are stored in the file answers_wiley.pdf.

1.7 EXERCISES

1. Store the values −20, −15, −5, 8, 12, 9, 2, 23, 19 in the R variable `x` and use the R command `sum` to verify that the sum of the values is 33.
2. For the data in Exercise 1, verify that the average is 3.67 using the R command `mean`.
3. What R commands can be used to compute an average without using the R command `mean`?
4. In Exercise 1, use R to sum the positive values ignoring the negative values.
5. In Exercise 1, use the `which` command to get the average of the values ignoring the largest value.
6. If the data in Exercise 1 are stored in the R variable `x`, speculate about the values corresponding to `x[abs(x)>=8 & x<8]`. Verify your speculation using this R command.
7. You record your commute time to work for 10 days, in minutes, and get 23, 18, 29, 22, 24, 27, 28, 19, 28, 23. Use R to determine the average, the shortest time, and the longest time.
8. Verify that the R commands

```
y=c(2,4,8)
z=c(1,5,2)
2*y
```

   return the values 4, 8, 16. Also, verify that the R command `y+z` returns 3, 9, 10 and that the command `y-2` returns 0, 2, 6.
9. Let `x = c(1, 8, 2, 6, 3, 8, 5, 5, 5, 5)`. Use R to compute the average using the `sum` and `length` commands. Next, use a single command to subtract the value 4 from each value stored in `x`. Finally, find the difference between the largest and smallest values stored in `x`. (This difference is called the range.) You can use the `max` and `min` functions or the `range` function.
10. For the data in Exercise 9, use R to subtract the average from each value, and then sum the results.

11. Imagine a matrix `m` having 100 rows and 2 columns. Further imagine that some of the values in the first column are NA, missing. Describe how the R function `is.na` can be used to eliminate the rows with missing values.
12. R has a built-in data set called ChickWeight. Verify that the R command

    ```
    mean(ChickWeight[,1])
    ```

    returns 121.8 but that the command

    ```
    mean(ChickWeight[,3])
    ```

    returns NA and a warning message even though the values in column 3 appear to be numeric. The reason for the warning message is that column 3 is stored as a factor variable. Arithmetic operations can only by performed on numeric or logical variables. Verify that

    ```
    mean(as.numeric(ChickWeight[,3]))
    ```

    returns 26.26.
13. Create a matrix with two rows and five columns with some of the entries stored as NA. Verify that the R function

    ```
    elimna(x)
    ```

    eliminates the rows with missing values. (This function is contained in the file Rallfun and can be installed as described in Section 1.4.)
14. R has a built-in data set called chickwts, which is stored in a data frame with two columns. (It differs from the built-in data set ChickWeight.) The first column contains the weight of chicks, and the second column indicates the type of feed they received, one of which is labeled horsebean. Use R to compute the average weight among chicks that were fed horsebean.
15. Let `x = c(1, 8, 2, 6, 3, 8, 5, 5, 5, 5)`. Describe two different R commands for summing the values in `x` ignoring the value 2 stored in `x[3]` and the value 3 stored in `x[5]`.
16. For the values used in the previous exercise, use two different R commands to sum all of the values not equal to 5.
17. For the data used in the previous two exercises, use a single R command to change all values equal to 8 to 7.
18. Create a matrix with four rows and two columns with the values 1, 2, 3, 4 and in the first column and 5, 6, 7, 8 in the second column.
19. Create a matrix with four columns and two rows with the values 1, 2, 3, 4 and in the first row and 11, 12, 13, 14 in the second row.