Getting Started With R 1

The purpose of this chapter is to introduce you to the R language and interpreter. After describing some of the basics of R, we will proceed to illustrate its use in a typical, if small, regression problem. We will then provide a brief description of R functions for familiar operations in basic statistics. The chapter concludes with an equally brief introduction to the *R Commander* graphical user interface (GUI) to R.

We know that many readers are in the habit of beginning a book at Chapter 1, skipping the Preface. The Preface to this *Companion* includes information about installing R and the **car** package on your computer. The **car** package, associated with the *R Companion to Applied Regression*, is necessary for many of the examples in the text. Moreover, the Preface includes information on the typographical and other conventions that we use in the text.

1.1 R Basics

Figure 1.1 shows the *RGui* (R *G*raphical *U*ser *I*nterface) for the Windows version of R. The most important element of the *Rgui* is the *R Console* window, which initially contains an opening message followed by a line with just a *command prompt*—the greater than (>) symbol. Interaction with R takes place at the command prompt. In Figure 1.1, we typed a simple command, `2 + 3`, followed by the Enter key. R interprets and executes the command, returning the value 5, followed by another command prompt. Figure 1.2 shows the similar *R.app* GUI for the Mac OS X version of R.

The menus in *RGui* and *R.app* provide access to many routine tasks, such as setting preferences, various editing functions, and accessing documentation. We draw your attention in particular to the *Packages* menu in the Windows *RGui* and to the *Packages & Data* menu in the Mac OS X *R.app*, both of which provide dialogs for installing and updating R packages. Unlike

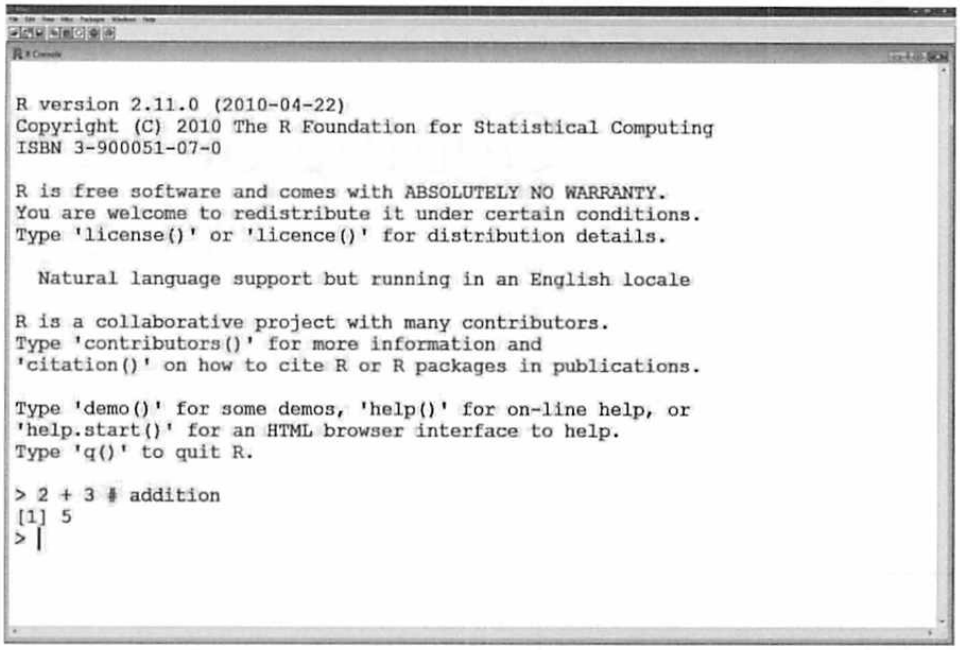

Figure 1.1 The *RGui* interface to the Windows version of R, shortly after the beginning of a session. This screen shot shows the default multiple-document interface (MDI); the single-document interface (SDI) looks similar but consists only of the *R Console* with the menu bar.

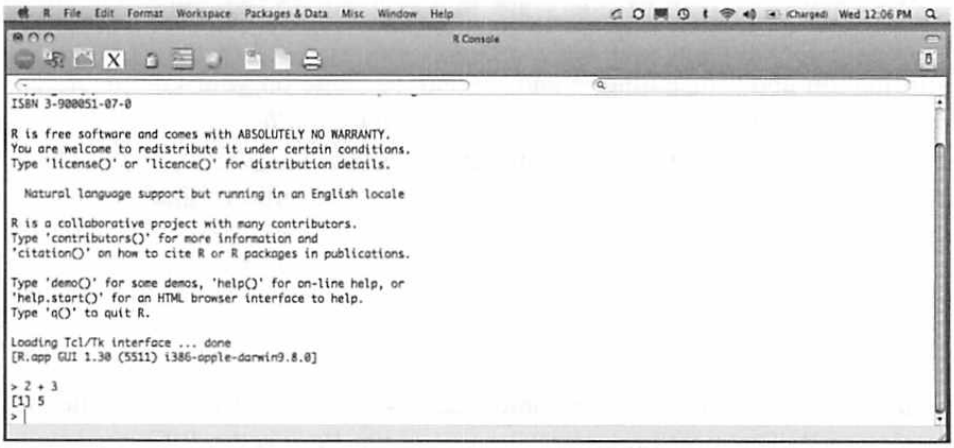

Figure 1.2 The *R.app* interface to the Mac OS X version of R.

many statistical analysis programs, the standard R menus do not provide direct access to the statistical functions in R, for which you will generally have to enter commands at the command prompt.

1.1.1 INTERACTING WITH THE INTERPRETER

Data analysis in R typically proceeds as an interactive dialogue with the interpreter. We type an R command at the > prompt, press the Enter key,

and the interpreter responds by executing the command and, as appropriate, returning a result, producing graphical output, or sending output to a file or device.

The R language includes the usual arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
^ or **	exponentiation

Here are some simple examples of arithmetic in R:

```
> 2 + 3 # addition

[1] 5

> 2 - 3 # subtraction

[1] -1

> 2*3   # multiplication

[1] 6

> 2/3   # division

[1] 0.6667

> 2^3   # exponentiation

[1] 8
```

Output lines are preceded by `[1]`. When the printed output consists of many values spread over several lines, each line begins with the index number of the first element in that line; an example will appear shortly. After the interpreter executes a command and returns a value, it waits for the next command, as signified by the > prompt. The pound sign (#) signifies a comment: Text to the right of # is ignored by the interpreter. We often take advantage of this feature to insert explanatory text to the right of commands, as in the examples above.

Several arithmetic operations may be combined to build up complex expressions:

```
> 4^2 - 3*2

[1] 10
```

In the usual notation, this command is $4^2 - 3 \times 2$. R uses standard conventions for precedence of mathematical operators. So, for example, exponentiation takes place before multiplication, which takes place before subtraction. If two operations have equal precedence, such as addition and subtraction, then they take place from left to right:

```
> 1 - 6 + 4

[1] -1
```

You can always explicitly specify the order of evaluation of an expression by using parentheses; thus, the expression `4^2 - 3*2` is equivalent to

```
> (4^2) - (3*2)

[1] 10
```

and

```
> (4 + 3)^2

[1] 49
```

is different from

```
> 4 + 3^2

[1] 13
```

Although spaces are not required to separate the elements of an arithmetic expression, judicious use of spaces can help clarify the meaning of the expression. Compare the following commands, for example:

```
> -2--3

[1] 1

> -2 - -3

[1] 1
```

Placing spaces around operators usually makes expressions more readable, as in the preceding examples. Readability of commands is generally improved by putting spaces around the binary arithmetic operators + and - but not around *, /, or ^.

1.1.2 R FUNCTIONS

In addition to the common arithmetic operators, R includes many—literally hundreds—of functions for mathematical operations, for statistical data analysis, for making graphs, and for other purposes. Function *arguments* are values passed to functions, and these are specified within parentheses after the function name. For example, to calculate the natural log of 100, that is $\log_e 100$ or ln 100, we type

```
> log(100)

[1] 4.605
```

To compute the log of 100 to the base 10, we specify

```
> log(100, base=10)

[1] 2

> log10(100) # equivalent

[1] 2
```

In general, arguments to R functions may be specified in the order in which they occur in the function definition or by the name of the argument followed by = (equals sign) and a value. In the command `log(100, base=10)`, the value `100` is matched to the first argument in the `log` function. The second argument, `base=10`, explicitly matches the value `10` to the argument `base`.

Different arguments are separated by commas, and for clarity, we prefer to leave a space after each comma, although these spaces are not required. Argument names may be abbreviated, as long as the abbreviation is unique; thus, the previous example may be rendered more compactly as

```
> log(100, b=10)

[1] 2
```

To obtain information about a function, use the `help` function. For example,

```
> help(log)
```

The result of executing this command is shown in abbreviated form in Figure 1.3, where the three widely separated dots (`...`) mean that we have elided some information. An alternative that requires less typing is to use the equivalent `?` (help) operator, `?log`.

Figure 1.3 is a typical R help page, giving first a brief description of the functions documented in the help page, followed by a listing of the available arguments, and then a discussion of the arguments. The `Details` and `Value` sections generally describe what the function does. All functions return a value, and the `log` function returns the logarithm of its first argument. Some functions, particularly those that draw graphs, don't *appear* to return a value and are used instead for the *side effect* of drawing a graph. Help pages usually include references to related functions, along with examples that you can execute to see how the documented functions work. Reading R documentation is an acquired skill, but once you become familiar with the form of the documentation, you will likely find the help pages very useful.

Help can be displayed in a help window as a plain-text file, or as an HTML page in a web browser. The HTML help format has several useful features, such as live hypertext links to other help pages, and is selected by the command `options(help_type="html")`. HTML help is the default option when R is installed.

The `help.start()` command opens a page in your web browser that gives direct access to a variety of resources, including HTML versions of the

```
log                    package:base                    R Documentation
Logarithms and Exponentials
Description:
     'log' computes logarithms, by default natural logarithms,
     'log10' computes common (i.e., base 10) logarithms, and
     'log2' computes binary (i.e., base 2) logarithms.  The general
     form 'log(x, base)' computes logarithms with base 'base'.
. . .
     'exp' computes the exponential function.
. . .
Usage:
     log(x, base = exp(1))
     logb(x, base = exp(1))
     log10(x)
     log2(x)
. . .
     exp(x)
. . .
Arguments:
       x: a numeric or complex vector.
    base: a positive or complex number: the base with respect to which
          logarithms are computed.  Defaults to e='exp(1)'.
Details:
     All except 'logb' are generic functions: methods can be
     defined for them individually or via the 'Math' group generic.
. . .
Value:
     A vector of the same length as 'x' containing the transformed
     values.  'log(0)' gives '-Inf', and negative values give
     'NaN'.
. . .
See Also:
     'Trig', 'sqrt', 'Arithmetic'.
Examples:
     log(exp(3))
     log10(1e7)# = 7
     x <- 10^-(1+2*1:9)
     cbind(x, log(1+x), log1p(x), exp(x)-1, expm1(x))
```

Figure 1.3 The documentation returned by the command `help(log)`. The ellipses (. . .) represent elided lines.

R manuals, hyperlinked help for all installed packages, a help search engine, frequently-asked-questions (FAQ) lists, and more. The `help.start()` command is so useful that you may want to put it into a startup file so that the help browser opens at the beginning of every R session (see the Preface and `?Startup`).

A novel feature of the R help system is the facility it provides to execute most examples in the help pages via the `example` command:

```
> example("log")

log> log(exp(3))
[1] 3

log> log10(1e7)  # = 7
```

```
[1] 7
. . .
```

The number `1e7` in the last example is given in scientific notation and represents $1 \times 10^7 = 10$ million.

A quick way to determine the arguments of many functions is to use the `args` function:

```
> args(log)

function (x, base = exp(1))
NULL
```

Because `base` is the second argument of the `log` function, we can also type

```
> log(100, 10)

[1] 2
```

specifying both arguments to the function (i.e., `x` and `base`) by position.

An argument to a function may have a *default* value—a value that the argument assumes if it is not explicitly specified in the function call. Defaults are shown in the function documentation and in the output of `args`. For example, the `base` argument to the `log` function defaults to `exp(1)` or $e^1 \approx 2.718$, the base of the natural logarithms.

R is largely a *functional programming language*, which means that both the standard programs that make up the language and the programs that users write are functions. Indeed, the distinction between standard and user-defined functions is somewhat artificial in R.[1] Even the arithmetic operators in R are really functions and may be used as such:

```
> '+'(2, 3)

[1] 5
```

We need to place back-ticks around `'+'` (single or double quotes also work) so that the interpreter does not get confused, but our ability to use `+` and the other arithmetic functions as *in-fix* operators, as in `2 + 3`, is really just syntactic "sugar," simplifying the construction of R expressions but not fundamentally altering the functional character of the language.

1.1.3 VECTORS AND VARIABLES

R would not be very convenient to use if we had to compute one value at a time. The arithmetic operators, and most R functions, can operate on more complex data structures than individual numbers. The simplest of these data structures is a numeric vector, or one-dimensional list of numbers.[2] In R an individual number is really a vector with a single element. A simple way to construct a vector is with the `c` function, which combines its elements:

[1] Section 1.1.6 briefly discusses user-defined functions; the topic is treated in greater depth in Chapter 8. Experienced programmers can also access programs written in Fortran and C from within R.

[2] We refer to vectors as "lists" using that term loosely, because *lists* in R are a distinct data structure (described in Section 2.3).

```
> c(1, 2, 3, 4)

[1] 1 2 3 4
```

Many other functions also return vectors as results. For example, the *sequence operator* (`:`) generates consecutive numbers, while the sequence function (`seq`) does much the same thing, but more flexibly:

```
> 1:4 # integer sequence

[1] 1 2 3 4

> 4:1

[1] 4 3 2 1

> -1:2

[1] -1  0  1  2

> seq(1, 4)

[1] 1 2 3 4

> seq(2, 8, by=2) # specify interval

[1] 2 4 6 8

> seq(0, 1, by=0.1) # non-integer sequence

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0

> seq(0, 1, length=11) # specify number of elements

 [1] 0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0
```

The standard arithmetic operators and functions apply to vectors on an element-wise basis:

```
> c(1, 2, 3, 4)/2

[1] 0.5 1.0 1.5 2.0

> c(1, 2, 3, 4)/c(4, 3, 2, 1)

[1] 0.2500 0.6667 1.5000 4.0000

> log(c(0.1, 1, 10, 100), 10)

[1] -1  0  1  2
```

If the operands are of different lengths, then the shorter of the two is extended by repetition, as in `c(1, 2, 3, 4)/2` above; if the length of the longer operand is not a multiple of the length of the shorter one, then a warning message is printed, but the interpreter proceeds with the operation, *recycling* the elements of the shorter operand:

```
> c(1, 2, 3, 4) + c(4, 3) # no warning

[1] 5 5 7 7

> c(1, 2, 3, 4) + c(4, 3, 2) # produces warning

[1] 5 5 5 8

Warning message:
In c(1, 2, 3, 4) + c(4, 3, 2) :
  longer object length is not a multiple of shorter object length
```

R would also be of little use if we were unable to save the results returned by functions; we do so by *assigning* values to *variables*, as in the following example:

```
> x <- c(1, 2, 3, 4) # assignment
> x # print

[1] 1 2 3 4
```

The left-pointing arrow (`<-`) is the *assignment operator*; it is composed of the two characters < (less than) and - (dash or minus), with no intervening blanks, and is usually read as *gets*: "The variable `x` *gets* the value `c(1, 2, 3, 4)`." The equals sign (=) may also be used for assignment in place of the arrow (`<-`), except inside a function call, where = is exclusively used to specify arguments by name. Because reserving the equals sign for specification of function arguments leads to clearer and less error-prone R code, we encourage you to use the arrow for assignment, even where = is allowed.[3]

As the preceding example illustrates, when the leftmost operation in a command is an assignment, nothing is printed. Typing the name of a variable, as in the second command immediately above, causes its value to be printed.

Variable names in R are composed of letters (`a–z`, `A–Z`), numerals (0–9), periods (`.`), and underscores (`_`), and they may be arbitrarily long. The first character must be a letter or a period, but variable names beginning with a period are reserved by convention for special purposes.[4] Names in R are case sensitive; so, for example, `x` and `X` are distinct variables. Using descriptive names, for example, `total.income` rather than `x2`, is almost always a good idea.

Three common naming styles are conventionally used in R: (1) separating the parts of a name by periods, as in `total.income`; (2) separating them by underscores, as in `total_income`; or (3) separating them by uppercase letters, termed *camel case*, as in `totalIncome`. For variable names, we prefer the first style, but this is purely a matter of taste.

R commands using defined variables simply substitute the value of the variable for its name:

[3] R also permits a right-pointing arrow for assignment, as in `2 + 3 -> x`.

[4] Nonstandard names may be used in a variety of contexts, including assignments, by enclosing the names in back-ticks, or in single or double quotes (e.g., `'first name' <- "John"`). In most circumstances, however, nonstandard names are best avoided.

```
> x/2

[1] 0.5 1.0 1.5 2.0

> (y <- sqrt(x))

[1] 1.000 1.414 1.732 2.000
```

In the last example, `sqrt` is the square-root function, and thus `sqrt(x)` is equivalent to `x^0.5`. To obtain printed output without having to type the name of the variable `y` as a separate command, we enclose the command in parentheses so that the assignment is no longer the leftmost operation. We will use this trick regularly.

Unlike in many programming languages, variables in R are dynamically defined. We need not tell the interpreter in advance how many values `x` is to hold or whether it contains integers (whole numbers), real numbers, character values, or something else. Moreover, if we wish, we may *redefine* the variable `x`:

```
(x <- rnorm(100))  # 100 standard normal random numbers

  [1]  0.58553  0.70947 -0.10930 -0.45350  0.60589 -1.81796  0.63010
  [8] -0.27618 -0.28416 -0.91932 -0.11625  1.81731  0.37063  0.52022
 [15] -0.75053  0.81690 -0.88636 -0.33158  1.12071  0.29872  0.77962
. . .
 [92] -0.85508  1.88695 -0.39182 -0.98063  0.68733 -0.50504  2.15772
 [99] -0.59980 -0.69455
```

The `rnorm` function generates standard-normal random numbers, in this case, 100 of them. Two additional arguments, not used in this example, allow us to sample values from a normal distribution with arbitrary mean and standard deviation; the defaults are `mean=0` and `sd=1`, and because we did not specify these arguments, the defaults were used. When a vector prints on more than one line, as in the last example, the index number of the leading element of each line is shown in square brackets.

The function `summary` is an example of a *generic function*: How it behaves depends on its argument. Applied as here to a numeric vector,

```
> summary(x)

   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
-2.380  -0.590   0.484   0.245   0.900   2.480
```

`summary` prints the minimum and maximum values of its argument, along with the mean, median, and first and third quartiles. Applied to another kind of object—a matrix, for example—`summary` gives different information, as we will see later.

1.1.4 NONNUMERIC VECTORS

Vectors may also contain nonnumeric values. For example,

```
> (words <- c("To", "be", "or", "not", "to", "be"))

[1] "To"  "be"  "or"  "not" "to"  "be"
```

is a *character vector* whose elements are character strings. There are R functions to work with character data. For example, to turn this vector into a single character string:

```
> paste(words, collapse=" ")

[1] "To be or not to be"
```

The very useful `paste` function pastes strings together (and is discussed, along with other functions for manipulating character data, in Section 2.4). The `collapse` argument, as its name implies, collapses the character vector into a single string, separating the elements with whatever is between the quotation marks, in this case one blank space.

A *logical vector* has all its elements either TRUE or FALSE:

```
> (vals <- c(TRUE, TRUE, FALSE, TRUE))

[1]  TRUE  TRUE FALSE  TRUE
```

The symbols T and F may also be used as logical values, but while TRUE and FALSE are *reserved symbols* in R, T and F are not, an omission that we regard as a design flaw in the language. For example, you can perniciously assign T <- FALSE and F <- TRUE (Socrates was executed for less!). For this reason, we suggest avoiding the symbols T and F.

Functions are available for working with logical vectors. For example, the ! operator negates a logical vector:

```
> !vals

[1] FALSE FALSE  TRUE FALSE
```

If we use logical values in arithmetic, R treats FALSE as if it were a zero and TRUE as if it were a one:

```
> sum(vals)

[1] 3
```

```
> sum(!vals)

[1] 1
```

More logical operators are described in the next section.

If we create a vector of mixed character strings, logical values, and numbers, we get back a vector of character strings:

```
> c("A", FALSE, 3.0)

[1] "A"     "FALSE" "3"
```

A vector of mixed numbers and logical values is treated as numeric, with FALSE becoming zero and TRUE becoming one. (Try it!) In the first case, we say that the logical and numeric values are *coerced* to character; in the second case, the logical values are coerced to numeric. In general, coercion in R takes place naturally and is designed to lose as little information as possible (see Section 2.6).

1.1.5 INDEXING VECTORS

If we wish to access—say, to print—only one of the elements of a vector, we can specify the index of the element within square brackets; for example, `x[12]` is the 12th element of the vector `x`:

```
> x[12]    # 12th element

[1] 1.817

> words[2] # second element

[1] "be"

> vals[3]     # third element

[1] FALSE
```

We may also specify a vector of indices:

```
> x[6:15] # elements 6 through 15

 [1] -1.8180  0.6301 -0.2762 -0.2842 -0.9193 -0.1162  1.8173  0.3706
 [9]  0.5202 -0.7505
```

Negative indices cause the corresponding values of the vector to be *omitted*:

```
> x[-(11:100)] # omit elements 11 through 100

 [1]  0.5855  0.7095 -0.1093 -0.4535  0.6059 -1.8180  0.6301 -0.2762
 [9] -0.2842 -0.9193
```

The parentheses around `11:100` serve to avoid generating numbers from −11 to 100, which would result in an error. (Try it!)

A vector can also be indexed by a logical vector of the same length. Logical values frequently arise through the use of *comparison operators*:

- == equals
- != not equals
- <= less than or equals
- < less than
- > greater than
- >= greater than or equals

The double-equals sign (==) is used for testing equality, because = is reserved for specifying function arguments and for assignment.

Logical values may also be used in conjunction with the *logical operators*:

- & and
- | or

Here are some simple examples:

```
> 1 == 2

[1] FALSE

> 1 != 2

[1] TRUE

> 1 <= 2

[1] TRUE

> 1 < 1:3

[1] FALSE  TRUE  TRUE

> 3:1 > 1:3

[1]  TRUE FALSE FALSE

> 3:1 >= 1:3

[1]  TRUE  TRUE FALSE

> TRUE & c(TRUE, FALSE)

[1]  TRUE FALSE

> c(TRUE, FALSE, FALSE) | c(TRUE, TRUE, FALSE)

[1]  TRUE  TRUE FALSE
```

A somewhat more extended example illustrates the use of the comparison and logical operators:

```
> (z <- x[1:10])

 [1]  0.5855  0.7095 -0.1093 -0.4535  0.6059 -1.8180  0.6301 -0.2762
 [9] -0.2842 -0.9193
```

```
> z < -0.5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
> z > 0.5
 [1]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
> z < -0.5 | z > 0.5  # < and > of higher precedence than |
 [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
> abs(z) > 0.5  # absolute value
 [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
> z[abs(z) > 0.5]
[1]  0.5855  0.7095  0.6059 -1.8180  0.6301 -0.9193
> z[!(abs(z) > 0.5)]
[1] -0.1093 -0.4535 -0.2762 -0.2842
```

The last of these commands uses the `!` operator, introduced in the last section, to negate the logical values returned by `abs(z) > 0.5` and thus returns the observations for which the condition is FALSE.

A few pointers about using these operators:

- We need to be careful in typing `z < -0.5`; although most spaces in R commands are optional, the space after < is crucial: `z <-0.5` would assign the value `0.5` to `z`. Even when spaces are not *required* around operators, they usually help to clarify R commands.
- Logical operators have lower precedence than comparison operators, and so `z < -0.5 | z > 0.5` is equivalent to `(z < -0.5) | (z > 0.5)`. When in doubt, parenthesize!
- The `abs` function returns the absolute value of its argument.
- As the last two commands illustrate, we can index a vector by a logical vector of the same length, selecting the elements with TRUE indices.

In addition to the vectorized *and* (&) and *or* (|) operators presented here, there are special *and* (&&) and *or* (||) operators that take individual logical values as arguments. These are often useful in writing programs (see Chapter 8).

1.1.6 USER-DEFINED FUNCTIONS

As you probably guessed, R includes functions for calculating many common statistical summaries, such as the mean of a vector:

```
> mean(x)
[1] 0.2452
```

Recall that x was previously defined to be a vector of 100 standard-normal random numbers. Were there no mean function, we could nevertheless have calculated the mean straightforwardly using sum and length:

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

[1] 0.2452
```

To do this repeatedly every time we need a mean would be inconvenient, and so in the absence of the standard R mean function, we could define our own mean function:

```
> myMean <- function(x) sum(x)/length(x)
```

- We define a function using the function function.[5] The arguments to function, here just x, are the *formal arguments* of the function being defined, myMean. As explained below, when the function myMean is called, an *actual argument* will appear in place of the formal argument. The remainder of the function definition is an R expression specifying the *body* of the function.
- The rule for naming functions is the same as for naming variables. We avoided using the name mean because we did not wish to *replace* the standard mean function, which is a generic function with greater utility than our simple version. For example, mean has an additional argument na.rm that tells R what to do if some of the elements of x are missing. We cannot overwrite the definitions of standard functions, but if we define a function of the same name, our version will be used in place of the standard function and is therefore said to *shadow* or *mask* the standard function. (This behavior is explained in Section 2.2.) In contrast to naming variables, in naming functions, we prefer using camel case (as in myMean) to separating words by periods (e.g., my.mean), because periods in function names play a special role in object-oriented programming in R (see Sections 1.4 and 8.7).
- The bodies of most user-defined functions are more complex than in this example, consisting of a *compound expression* comprising several simple R expressions, enclosed in braces and separated by semicolons or new-lines. We will introduce additional information about writing functions as required and take up the topic more systematically in Chapter 8.

Having defined the function myMean, we may use it in the same manner as the standard R functions. Indeed, most of the standard functions in R are themselves written in the R language.[6]

[5] We could not resist writing that sentence! Actually, however, function is a *special form*, not a true function, because its arguments (here, the formal argument x) are not evaluated. The distinction is technical, and it will do no harm to think of function as a function that returns a function as its result.

[6] Some of the standard R functions are *primitives*, in the sense that they are defined in code written in the lower-level languages C and Fortran.

```
> myMean(x)

[1] 0.2452

> y # from sqrt(c(1, 2, 3, 4))

[1] 1.000 1.414 1.732 2.000

> myMean(y)

[1] 1.537

> myMean(1:100)

[1] 50.5

> myMean(sqrt(1:100))

[1] 6.715
```

As these examples illustrate, there is no necessary correspondence between the name of the formal argument `x` of the function `myMean` and the actual argument to the function. Function arguments are evaluated by the interpreter, and it is the *value* of the argument that is passed to the function, not its name. Thus, in the last of the three examples above, the function call `sqrt(1:100)` must first be evaluated, and then the result is used as the argument to `myMean`. Function arguments, along with variables that are defined within a function, are *local* to the function: Local variables exist only while the function executes and are distinct from *global* variables of the same name. For example, the last call to `myMean` passed the value of `sqrt(1:100)` (i.e., the square roots of the integers from 1 to 100) to the argument `x`, but this argument did not change the contents of the global variable `x` (see p. 10):

```
> x

  [1]  0.58553  0.70947 -0.10930 -0.45350  0.60589 -1.81796  0.63010
. . .
 [99] -0.59980 -0.69455
```

1.1.7 COMMAND EDITING AND OUTPUT MANAGEMENT

In the course of typing an R command, you may find it necessary to correct or modify the command before pressing Enter. The Windows *R Console* supports command-line editing:[7]

- You can move the cursor with the left and right arrow, Home, and End keys.
- The Delete key deletes the character under the cursor.
- The Backspace key deletes the character to the left of the cursor.
- The standard Windows *Edit* menu and keyboard shortcuts may be employed, along with the mouse, to block, copy, and paste text.

[7]The menu selection *Help* → *Console* will display these hints.

- In addition, R implements a command-history mechanism that allows you to recall and edit previously entered commands without having to retype them. Use the up and down arrow keys to move backward and forward in the command history. Press Enter in the normal manner to submit a recalled, and possibly edited, command to the interpreter.

The Mac OS X *R.app* behaves similarly, and somewhat more flexibly, in conformity with the usual OS X conventions.

Writing all but the simplest functions directly at the command prompt is impractical and possibly frustrating, and so using a programming editor with R is a good idea. Both the Windows and Mac OS X implementations of R include basic programming or script editors. We recommend that new users of R use these basic editors before trying a more sophisticated programming editor. You can open a new R script in the Windows *RGui* via the *File* → *New script* menu, or an existing script file via *File* → *Open script*. Similar *New Document* and *Open Document* selections are available under the Mac OS X *R.app File* menu. By convention, R script files have names that end with the extension or file type .R—for example, mycommands.R.

We also strongly recommend the use of an editor for data analysis in R, typing commands into the editor and then submitting them for execution rather than typing them directly at the command prompt. Using an editor simplifies finding and fixing errors, especially in multiline commands, and facilitates trying variations on commands. Moreover, when you work in the editor, you build a permanent, reusable record of input to your R session as a by-product.

Using the script editor in the Windows version of R, simply type commands into the editor, select them with the mouse, and then select *Edit* → *Run line or selection* or press the key combination Control-R to send the commands to the *R Console*. The procedure is similar in Mac OS X, except that commands are sent to the R interpreter by pressing the key combination command-return.

As you work, you can save text and graphical output from R in a word-processor (e.g., Microsoft Word or OpenOffice Writer) document. Simply block and copy the text output from the *R Console* and paste it into the word-processor document, taking care to use a monospaced (i.e., typewriter) font, such as `Courier New`, so that the output lines up properly. Word processors, however, make poor programming editors, and we recommend against their use for composing scripts of R commands.

Similarly, under Windows, you can copy and paste graphs: Right-clicking on a graphics window brings up a context menu that allows you to save the graph to a file or copy it to the Windows clipboard, from which it can be pasted into a word-processor document, for example. Alternatively, you can use the graphics window's *File* menu to save a graph. Copying the graph to the clipboard as a Windows Metafile rather than as a bitmap generally produces a more satisfactory result. Using *R.app* under Mac OS X, you can save the current plot in a *Quartz* graphics device window via the *File* → *Save as*

menu, which by default saves a PDF file containing the graph; you can then import the PDF file into a word-processor document.[8]

For LaTeX users, R supports a sophisticated system called Sweave for interleaving text and graphics with executable R code (for details, see Leisch, 2002, 2003). Indeed, we used Sweave to write this book!

1.1.8 WHEN THINGS GO WRONG

No one is perfect, and it is impossible to use a computer without making mistakes. Part of the craft of computing is learning to recognize the source of errors. We hope that the following advice and information will help you fix errors in R commands:

- Although it never hurts to be careful, do not worry too much about generating errors. An advantage of working in an interactive system is that you can proceed step by step, fixing mistakes as you go. R is also unusually forgiving in that it is designed to restore the workspace to its previous state when a command results in an error.
- If you are unsure whether a command is properly formulated or whether it will do what you intend, try it out and carefully examine the result. You can often debug commands by trying them on a scaled-down problem with an obvious answer. If the answer that you get differs from the one that you expected, focus your attention on the nature of the difference. Similarly, reworking examples from this *Companion*, from R help pages, or from textbooks or journal articles can help convince you that your programs are working properly.[9]
- When you do generate an error, don't panic! Read the error or warning message carefully. Although some R error messages are cryptic, others are informative, and it is often possible to figure out the source of the error from the message. Some of the most common errors are merely typing mistakes. For example, when the interpreter tells you that an object is not found, suspect a typing error, or that you have forgotten to load the package or read the file containing the object (e.g., a function).
- Sometimes, however, the source of an error may be subtle, particularly because an R command can generate a sequence of function calls of one function by another, and the error message may originate deep within this sequence. The `traceback` function, called with no arguments, provides information about the sequence of function calls leading up to an error. To create a simple example, we begin by writing a function to compute the variance of a variable, checking the output against the standard `var` function:

```
> myVar <- function(x) sum((x - myMean(x))^2)/(length(x) - 1)
> myVar(1:100)
```

[8] See Section 7.4 for more information on handling graphics devices in R.

[9] Sometimes, however, this testing may convince you that the published results are wrong, but that is another story.

```
[1] 841.7

> var(1:100) # check

[1] 841.7
```

We deliberately produce an error by foolishly calling myVar with a nonnumeric argument:

```
> letters

 [1] "a" "b" "c" "d" "e" "f" "g" "h" "i" "j" "k" "l" "m" "n" "o" "p"
[17] "q" "r" "s" "t" "u" "v" "w" "x" "y" "z"

> myVar(letters)

Error in sum(x) : invalid 'type' (character) of argument
```

The built-in variable `letters` contains the lowercase letters, and of course, calculating the variance of character data makes no sense. Although the source of the problem is obvious, the error occurs in the sum function, not directly in myVar; `traceback` shows the sequence of function calls culminating in the error:

```
> traceback()

2: myMean(x)
1: myVar(letters)
```

- Not all errors generate error messages. Indeed, the ones that do not are more pernicious, because you may fail to notice them. Always check your output for reasonableness, and follow up suspicious results.
- If you need to interrupt the execution of a command, you may do so by pressing the Esc (escape) key, by using the mouse to press the *Stop* button in the toolbar, or (under Windows) by selecting the *Misc* → *Stop current computation* menu item.
- There is much more information on debugging R code in Section 8.6.1.

1.1.9 GETTING HELP AND INFORMATION

We have already explained how to use the `help` function and `?` operator to get information about an R function. But what do you do if this information is insufficient or if you don't know the name of the function that you want to use? You may not even know whether a function to perform a specific task *exists* in the standard R distribution or in one of the contributed packages on CRAN. This is not an insignificant problem, for there are hundreds of functions in the standard R packages and literally thousands of functions in the more than 2,500 packages on CRAN.

Although there is no completely adequate solution to this problem, there are several R resources beyond `help` and `?` that can be of assistance:[10]

[10] In addition, we have already introduced the `help.start` command, and in Section 4.9, we describe the use of the `hints` function in the **hints** package to obtain information about functions that can be used with a particular R object.

- The `apropos` command searches for currently accessible objects whose names contain a particular character string. For example,

```
> apropos("log")

. . .
 [7] "dlogis"                "is.logical"
 [9] "log"                   "log10"
[11] "log1p"                 "log2"
[13] "logb"                  "Logic"
[15] "logical"               "logLik"
. . .
```

- Casting a broader net, the `help.search` command searches the titles and certain other fields in the help files of all R packages installed on your system, showing the results in a pop-up window. For example, try the command `help.search("loglinear")` to find functions related to loglinear models (discussed in Section 5.6). The `??` operator is a synonym for `help.search`—for example, `??loglinear`.
- If you have an active Internet connection, you can search even more broadly with the `RSiteSearch` function. For example, to look in all standard and CRAN packages—even those not installed on your system—for functions related to loglinear models, you can issue the command `RSiteSearch("loglinear", restrict="functions")`. The results appear in a web browser. See `?RSiteSearch` for details.
- The CRAN *task views* are documents that describe facilities in R for applications in specific areas such as Bayesian statistics, econometrics, psychometrics, social statistics, and spatial statistics. The approximately two-dozen task views are available via the command `carWeb("taskviews")`, which uses the `carWeb` function from the **car** package, or directly by pointing your browser at `http://cran.r-project.org/web/views/`.
- The command `help(package="`*`package-name`*`")`—for example, `help(package="car")`—shows information about an installed package, such as an index of help topics documented in the package.
- Some packages contain *vignettes*, discursive documents describing the use of the package. To find out what vignettes are available in the packages installed on your system, enter the command `vignette()`. The command `vignette(package="`*`package-name`*`")` displays the vignettes available in a particular installed package, and the command `vignette("`*`vignette-name`*`")` or `vignette("`*`vignette-name`*`", package="`*`package-name`*`")` opens a specific vignette.
- The *Help* menu in the Mac OS X and Windows versions of R provides self-explanatory menu items to access help pages, online manuals, the `apropos` function, and links to the R websites.
- As you might expect, help on R is available on the Internet from a wide variety of sources. The website `www.rseek.org` provides a custom Google search engine specifically designed to look for R-related

documents (try searching for **car** using this search site). The page `www.r-project.org/search.html` lists other possibilities for web searching.

- Finally, *Rhelp* is a very active email list devoted to answering users' questions about R, and there are also several more specialized R mailing lists (see `www.r-project.org/mail.html`). Before posting a question to *Rhelp* or to one of the other email lists, however, *please carefully read the posting guide* at `www.r-project.org/posting-guide.html`.

1.1.10 CLEANING UP

User-defined variables and functions exist in R in a region of memory called the *workspace*. The R workspace can be saved at the end of a session or even during the session, in which case it is automatically loaded at the start of the next session. Different workspaces can be saved in different directories, as a means of keeping projects separate. Starting R in a directory loads the corresponding workspace.[11]

The `objects` function lists the names of variables and functions residing in the R workspace:

```
> objects()

[1] "myMean"    "myVar"     "vals"      "words"     "x"
[6] "y"         "z"
```

The function `objects` requires no arguments, but we nevertheless need to type parentheses after the function name. Were we to type only the name of the function, then `objects` would not be called—instead the *definition* of the `objects` function would be printed. (Try it!) This is an instance of the general rule that entering the name of an R object—in this case, the function `objects`—causes the object to be printed.

It is natural in the process of using R to define variables—and occasionally functions—that we do not want to retain. It is good general practice in R, especially if you intend to save the workspace, to clean up after yourself from time to time. To this end, we use the `remove` function to delete the variables `x`, `y`, `z`, `vals` and `words`:

```
> remove(x, y, z, vals, words)

> objects()

[1] "myMean"    "myVar"
```

We keep the functions `myMean` and `myVar`, pretending that we still intend to use them.

[11] See the R documentation for additional information on organizing separate projects.

1.1.11 ENDING THE R SESSION

The function `quit` or its equivalent, `q`, is used to exit from R:

```
> quit()

Save workspace image? [y/n/c]:
```

Answering `y` will save the workspace in the current directory, an operation that we generally do not recommend;[12] use `n` to avoid saving the workspace or `c` to cancel quitting. Entering `quit(save="n")` suppresses the question. You can also exit from R via the *File* menu or by clicking on the standard close-window button—the red button at the upper right in Windows and the upper left in Mac OS X.

1.2 An Extended Illustration: Duncan's Occupational-Prestige Regression

In this section, we illustrate how to read data from a file into an R *data frame* (data set), how to draw graphs to examine the data using both standard R functions and some of the specialized functions included in the **car** package, how to perform a linear least-squares regression analysis, and how to check the adequacy of the preliminary regression model using a variety of diagnostic methods. It is our intention both to introduce some of the capabilities of R and to convey the flavor of using R for statistical data analysis. All these topics are treated at length later in the book, so you should not be concerned if you don't understand all the details.

The data in the file Duncan.txt were originally analyzed by Duncan (1961).[13] The first few lines of the data file are as follows:

```
                type income education  prestige
accountant      prof 62      86        82
pilot           prof 72      76        83
architect       prof 75      92        90
author          prof 55      90        76
chemist         prof 64      86        90
minister        prof 21      84        87
professor       prof 64      93        93
dentist         prof 80     100        90
```

[12] A saved workspace will be loaded automatically in a subsequent session, a situation that often results in confusion, in our experience, especially among new users of R. We therefore recommend that you start each R session with a pristine workspace and instead save the script of the commands you use during a session that you may wish to recover (see the discussion of programming editors in Section 1.1.7). Objects can then conveniently be re-created as needed by executing the commands in the saved script. Admittedly, whether to save workspaces or scripts of commands is partly a matter of preference and habit.

[13] The Duncan.txt file, along with the other files used in this text, are available on the website for this *Companion*, at the web address given in the Preface. To reproduce the example, download the data file to a convenient location on your hard disk. Alternatively, you can open a copy of the file in your web browser with the command `carWeb(data="Duncan.txt")` and then save it to your disk.