1

Getting Started With R and RStudio

This chapter shows you how to conduct statistical data analysis using R as the primary data analysis tool and RStudio as the primary tool for organizing your data analysis workflow and for producing reports. The goal is to conduct *reproducible research*, in which the finished product not only summarizes your findings but also contains all of the instructions and data needed to replicate your work. Consequently, you, or another skilled person, can easily understand what you have done and, if necessary, reproduce it. If you are a student using R for assignments, you will simultaneously analyze data using R, make your R commands available to your instructor, and write up your findings using R Markdown in RStudio. If you are a researcher or a data analyst, you can follow the same general process, preparing documents that contain reproducible results ready for distribution to others.

- We begin the chapter in Section 1.1 with a discussion of RStudio *projects*, which are a simple, yet powerful, means of organizing your work that takes advantage of the design of RStudio.
- We then introduce a variety of basic features of R in Section 1.2, showing you how to use the *Console* in RStudio to interact directly with the R interpreter; how to call R functions to perform computations; how to work with vectors, which are one-dimensional arrays of numbers, character strings, or logical values; how to define variables; and how to define functions.
- In Section 1.3, we explain how to locate and fix errors and how to get help with R.
- Direct interaction with R is ocassionally useful, but in Section 1.4 we show you how to work more effectively by using the RStudio editor to create scripts of R commands, making it easier to correct mistakes and to save a permanent record of your work. We then outline a more sophisticated and powerful approach to reproducible research, combining R commands with largely free-form explanatory material in an R Markdown document, which you can easily convert into a neatly typeset report.

- Although the focus of the *R Companion* is on using R for regression modeling, Section 1.6 enumerates some generally useful R functions for basic statistical methods.
- Finally, Section 1.7 explains how so-called *generic functions* in R are able to adapt their behavior to different kinds of data, so that, for example, the `summary()` function produces very different reports for a data set and for a linear regression model.

By the end of the chapter, you should be able to start working efficiently in R and RStudio.

We know that many readers are in the habit of beginning a book at Chapter 1, skipping the Preface. The Preface to this book, however, includes information about installing R and RStudio on your computer, along with the **car**, **effects**, and **carData** packages, which are associated with the *R Companion to Applied Regression* and are necessary for many of the examples in the text. We suggest that you rework the examples as you go along, because data analysis is best learned by doing, not simply by reading. Moreover, the Preface includes information on the typographical and other conventions that we use in the text. So, if you haven't yet read the Preface, please back up and do so now!

1.1 Projects in RStudio

Projects are a means of organizing your work so that both you and RStudio can keep track of all the files relevant to a particular activity. For example, a student may want to use R in several different projects, such as a statistics course, a sociology course, and a thesis research project. The data and data analysis files for each of these activities typically differ from the files associated with the other activities. Similarly, researchers and data analysts generally engage in several distinct research projects, both simultaneously and over time. Using RStudio projects to organize your work will keep the files for distinct activities separate from each other.

If you return to a project after some time has elapsed, you can therefore be sure that all the files in the project directory are relevant to the work you want to reproduce or continue, and if you have the discipline to put all files that concern a project in the project directory, everything you need should be easy to find. By organizing your work in separate project directories, you are on your way to conducting fully reproducible research.

Your first task is to create a new project directory. You can keep a project directory on your hard disk or on a flash drive. Some cloud services, like Dropbox, can also be used for saving a project directory.[1] To create a project, select *File > New Project* from the RStudio menus. In the resulting sequence of dialog boxes, successively select *Existing Directory* or *New Directory*, depending on whether or not the project directory already exists; navigate by pressing the *Browse...* button to the location

[1] At present, Google Drive appears to be incompatible with RStudio projects.

where you wish to create the project; and, for a new directory, supply the name of the directory to be created. We assume here that you enter the name R-Companion for the project directory, which will contain files relevant to this book. You can use the R-Companion project as you work through this and other chapters of the *R Companion*.

Creating the R-Companion project changes the RStudio window, which was depicted in its original state in Figures 2 and 3 in the Preface (pages xix and xx), in two ways: First, the name of the project you created is shown in the upper-right corner of the window. Clicking on the project name displays a drop-down menu that allows you to navigate to other projects you have created or to create a new project. The second change is in the *Files* tab, located in the lower-right pane, which lists the files in your project directory. If you just created the R-Companion project in a new directory, you will only see one file, R-Companion.Rproj, which RStudio uses to administer the project.

Although we don't need them quite yet, we will add a few files to the R-Companion project, typical of the kinds of files you might find in a project directory. Assuming that you are connected to the internet and have previously installed the **car** package, type the following two commands at the > command prompt in the *Console* pane,[2] remembering to press the Enter or return key after each command:

```
library("car")

 Loading required package: carData

carWeb(setup=TRUE)
```

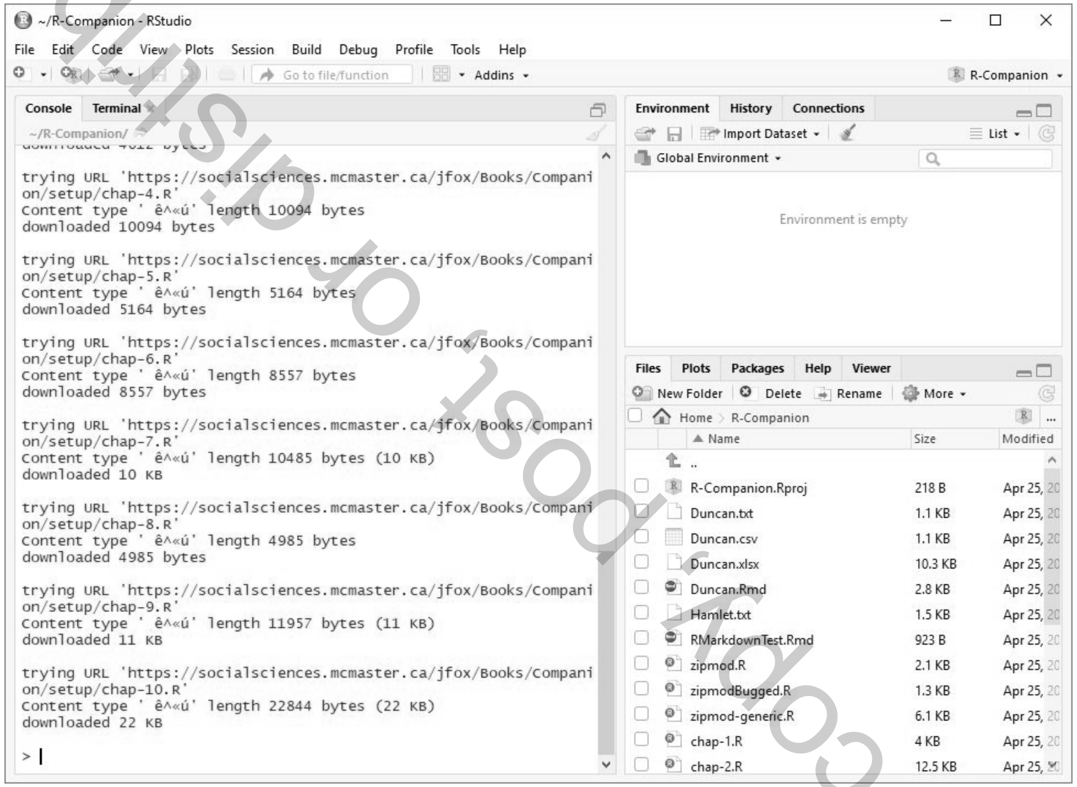
The first of these commands loads the **car** package.[3] When **car** is loaded, the **carData** package is automatically loaded as well, as reflected in the message printed by the `library()` command. In subsequent chapters, we suppress package-startup messages to conserve space. The second command calls a function called `carWeb()` in the **car** package to initialize your project directory by downloading several files from the website for this book to the R-Companion project directory. As shown in Figure 1.1, information about the downloaded files is displayed in the *Console*, and the files are now listed in the *Files* tab. We will use these files in this chapter and in a few succeeding chapters.

Files in an RStudio project typically include plain-text data files, usually of file type .txt or .csv, files of R commands, of file type .R or .r, and R Markdown files, of file type .Rmd, among others. In a complex project, you may wish to create subdirectories of the main project directory, for example, a Data subdirectory for data files or a Reports subdirectory for R Markdown files.

RStudio starts in a "default" working directory whenever you choose not to use a project. You can reset the default directory, which is initially set to your

[2] As we explained in the Preface, we don't show the command prompt when we display R input in the text.

[3] If you haven't installed the **car** package, or if the `library("car")` command produces an error, then please (re)read the Preface to get going.



```
trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-4.R'
Content type ' ê^«ú' length 10094 bytes
downloaded 10094 bytes

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-5.R'
Content type ' ê^«ú' length 5164 bytes
downloaded 5164 bytes

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-6.R'
Content type ' ê^«ú' length 8557 bytes
downloaded 8557 bytes

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-7.R'
Content type ' ê^«ú' length 10485 bytes (10 KB)
downloaded 10 KB

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-8.R'
Content type ' ê^«ú' length 4985 bytes
downloaded 4985 bytes

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-9.R'
Content type ' ê^«ú' length 11957 bytes (11 KB)
downloaded 11 KB

trying URL 'https://socialsciences.mcmaster.ca/jfox/Books/Compani
on/setup/chap-10.R'
Content type ' ê^«ú' length 22844 bytes (22 KB)
downloaded 22 KB

> |
```

Name	Size	Modified
..		
R-Companion.Rproj	218 B	Apr 25, 20
Duncan.txt	1.1 KB	Apr 25, 20
Duncan.csv	1.1 KB	Apr 25, 20
Duncan.xlsx	10.3 KB	Apr 25, 20
Duncan.Rmd	2.8 KB	Apr 25, 20
Hamlet.txt	1.5 KB	Apr 25, 20
RMarkdownTest.Rmd	923 B	Apr 25, 20
zipmod.R	2.1 KB	Apr 25, 20
zipmodBugged.R	1.3 KB	Apr 25, 20
zipmod-generic.R	6.1 KB	Apr 25, 20
chap-1.R	4 KB	Apr 25, 20
chap-2.R	12.5 KB	Apr 25,

Figure 1.1 RStudio window for the R-companion project after adding the files used in this and subsequent chapters.

Documents directory in Windows or your home directory in macOS, by selecting *Tools* > *Global Options* from the RStudio menus, and then clicking the *Browse...* button on the *General* tab, navigating to the desired location in your file system. We find that we almost never work in the default directory, however, because creating a new RStudio project for each problem is very easy and because putting unrelated files in the same directory is a recipe for confusion.

1.2 R Basics

1.2.1 Interacting With R Through the Console

Data analysis in R typically proceeds as an interactive dialogue with the interpreter, which may be accessed directly in the RStudio *Console* pane. We can type an R command at the > *prompt* in the *Console* (which is not shown in the R input displayed below) and press the Enter or return key. 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] 0.66667

2^3   # exponentiation

 [1] 8
```

Output lines are preceded by [1]. When the printed output consists of many values (a "vector": see Section 1.2.4) 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 indicated by the > prompt. The pound sign or hash mark (#) signifies a *comment*, and 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 mathematical 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 are evaluated 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, and some style standards for R code suggest that they *always* be used. We feel, however, that readability of commands is generally improved by putting spaces around the binary arithmetic operators + and - but not usually around `*`, `/`, or `^`.

Interacting directly with the R interpreter by typing at the command prompt is, for a variety of reasons, a poor way to organize your work. In Section 1.4, we'll show you how to work more effectively using scripts of R commands and, even better, dynamic R Markdown documents.

1.2.2 Editing R Commands in the Console

- The arrow keys on your keyboard are useful for navigating among commands previously entered into the *Console*: Use the ↑ and ↓ keys to move up and down in the the command history.
- You can also access the command history in the RStudio *History* tab: Double-clicking on a command in the *History* tab transfers the command to the > prompt in the *Console*.
- The command shown after the > prompt can either be re-executed or edited. Use the → and ← keys to move the cursor within the command after the >. Use the backspace or delete key to erase a character. Typed characters will be inserted at the cursor.
- You can also move the cursor with the mouse, left-clicking at the desired point.

1.2.3 R Functions

In addition to the common arithmetic operators, the packages in the standard R distribution include hundreds of functions (programs) for mathematical operations, for manipulating data, for statistical data analysis, for making graphs, for working with files, 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 use the `log()` function:[4]

[4] Here's a quick review of logarithms ("logs"), which play an important role in statistical data analysis (see, e.g., Section 3.4.1):

- The log of a positive number x to the base b (where b is also a positive number), written $\log_b x$, is the exponent to which b must be raised to produce x. That is, if $y = \log_b x$, then $b^y = x$.
- Thus, for example, $\log_{10} 100 = 2$ because $10^2 = 100$; $\log_{10} 0.01 = -2$ because $10^{-2} = 1/10^2 = 0.01$; $\log_2 8 = 3$ because $2^3 = 8$; and $\log_2 1/8 = -3$ because $2^{-3} = 1/8$.
- So-called *natural logs* use the base $e \approx 2.71828$.
- Thus, for example, $\log_e e = 1$ because $e^1 = e$.
- Logs to the bases 2 and 10 are often used in data analysis, because powers of 2 and 10 are familiar. Logs to the base 10 are called "common logs."
- Regardless of the base b, $\log_b 1 = 0$, because $b^0 = 1$.
- Regardless of the base, the log of zero is undefined (or taken as $\log 0 = -\infty$), and the logs of negative numbers are undefined.

```
log(100)

  [1] 4.6052
```

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

```
log(100, base=10)

  [1] 2
```

We could equivalently use the specialized `log10()` function:

```
log10(100) # equivalent

  [1] 2
```

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 = (the equals sign) and a value. In the command `log(100, base=10)`, the value `100` is implicitly matched to the first argument of the `log` function. The second argument in the function call, `base=10`, explicitly matches the value `10` to the argument `base`. Arguments specified by name need not appear in a function call in the same order that they appear in the function definition.

Different arguments are separated by commas, and, for clarity, we prefer to leave a space after each comma, although these spaces are not required. Some stylistic standards for R code recommend placing spaces around = in assigning values to arguments, but we usually find it clearer not to insert extra spaces here. Function-argument names may be abbreviated, as long as the abbreviation is unique; thus, the previous example may be rendered more compactly as

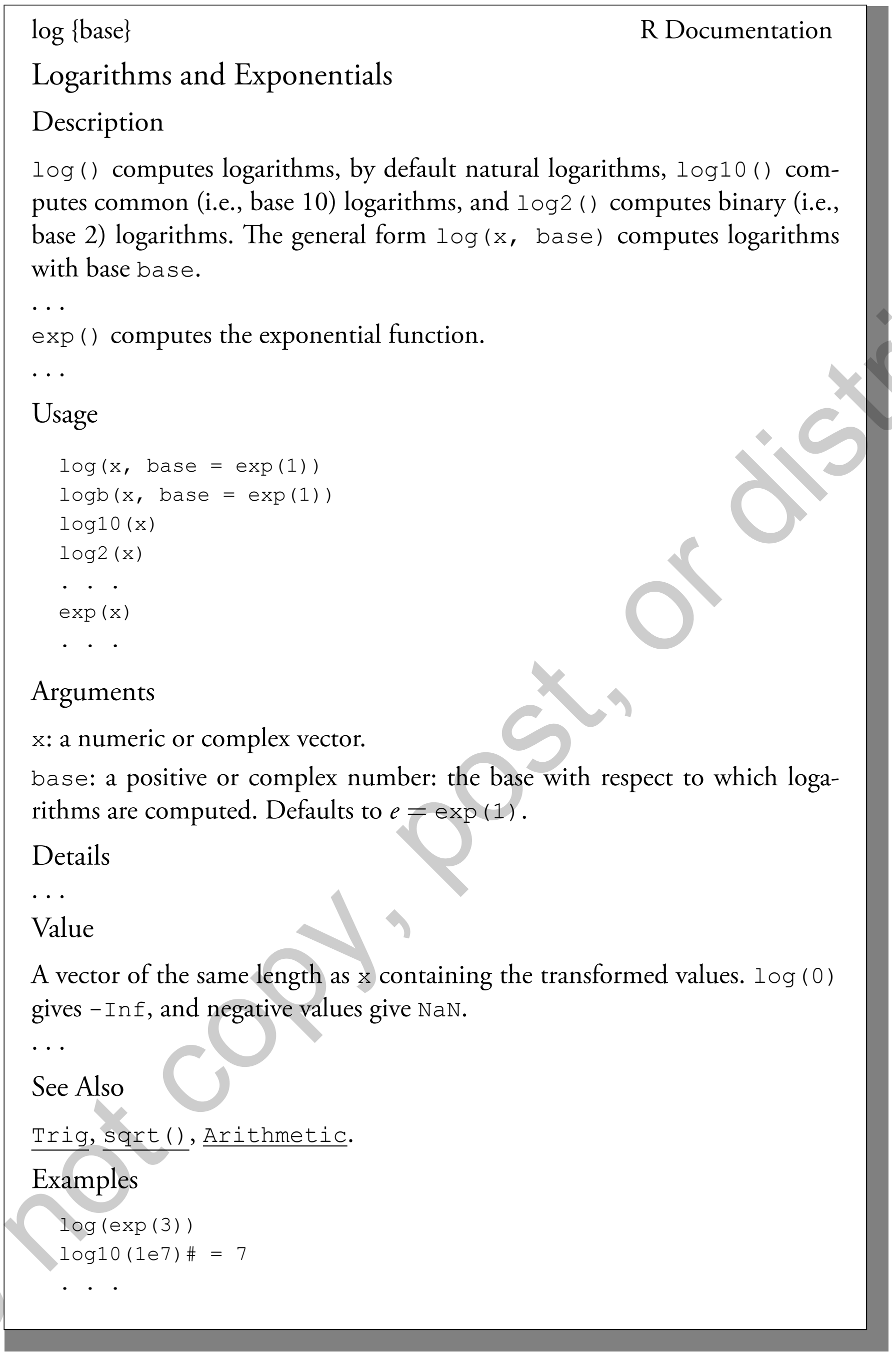
```
log(100, b=10)

  [1] 2
```

because the `log()` function has only one argument beginning with the letter "`b`." We generally prefer *not* to abbreviate function arguments because abbreviation promotes unclarity.

This example begs a question, however: How do we know what the arguments to the `log()` function are? To obtain information about a function, use the `help()` function or, synonymously, the `?` help operator. For example,

```
help("log")
?log
```

Either of these equivalent commands opens the R *help page* for the `log()` function and some closely associated functions, such as the exponential function, `exp(x)` $= e^x$, in the RStudio *Help* tab. Figure 1.2 shows the resulting help page in abbreviated form, where three widely separated dots (`. . .`) mean that we have elided some information, a convention that we'll occasionally use to abbreviate R output as well.



log {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

. . .

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
  . . .
```

Figure 1.2 Abbreviated documentation displayed by the command `help("log")`. The ellipses (. . .) represent elided lines, and the underscored text under “See Also” indicates hyperlinks to other help pages—click on a link to go to the corresponding help page. The symbol `-Inf` in the “Value” section represents minus infinity ($-\infty$), and `NaN` means “not a number.”

The `log()` help page is more or less typical of help pages for R functions in both standard R packages and in contributed packages obtained from CRAN. The *Description* section of the help page provides a brief, general description of the documented functions; the *Usage* section shows each documented function, its arguments, and argument default values (for arguments that have defaults, see below); the *Arguments* section explains each argument; the *Details* section (suppressed in Figure 1.2) elaborates the description of the documented functions; the *Value* section describes the value returned by each documented function; the *See Also* section includes hyperlinked references to other help pages; and the *Examples* section illustrates the use of the documented functions. There may be other sections as well; for example, help pages for functions documented in contributed CRAN packages typically have an *Author* section.

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 second example is given in *scientific notation* and represents $1 \times 10^7 = 10$ million. Scientific notation may also be used in R output to represent very large or very small numbers.

A quick way to determine the arguments of an R function is to use the `args()` function:[5]

```
args("log")

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

Because `base` is the second argument of the `log()` function, to compute $\log_{10} 100$, 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 that is used if the argument is not explicitly specified in a 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 $e^1 \approx 2.71828$, the base of the natural logarithms.

[5] Disregard the `NULL` value returned by `args()`.

1.2.4 Vectors and Variables

R would not be very convenient to use if we always 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 array of numbers.[6] An individual number in R is really a vector with a single element.

A simple way to construct a vector is with the `c()` function, which combines its elements:

```
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 whole 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 # descending

  [1] 4 3 2 1

-1:2 # negative to positive

  [1] -1  0  1  2

seq(1, 4) # equivalent to 1:4

  [1] 1 2 3 4

seq(2, 8, by=2) # specify interval between elements

  [1] 2 4 6 8

seq(0, 1, by=0.1) # noninteger 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 extend to vectors in a natural manner on an elementwise 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] 0.25000 0.66667 1.50000 4.00000

log(c(0.1, 1, 10, 100), base=10)

  [1] -1  0  1  2
```

[6] We refer here to vectors informally as one-dimensional "arrays" using that term loosely, because *arrays* in R are a distinct data structure (described in Section 2.4).

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, where the `2` in the denominator is effectively repeated four times. 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 be of little practical use if we were unable to save the results returned by functions to use them in further computation. A value is saved by *assigning* it to a *variable*, as in the following example, which assigns the vector `c(1, 2, 3, 4)` to the variable `x`:

```
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 used exclusively to specify arguments by name. We generally recommend the use of the arrow for assignment.[7]

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 above is equivalent to typing the command `print(x)` and causes the value of `x` to be printed.

Variable and function names in R are composed of letters (`a`–`z`, `A`–`Z`), numerals (`0`–`9`), periods (`.`), and underscores (`_`), and they may be arbitrarily long. In particular, the symbols `#` and `-` should not appear in variable or function names. The first character in a name must be a letter or a period, but variable names beginning with a period are reserved by convention for special purposes.[8] Names in R are case sensitive: So, for example, `x` and `X` are distinct variables. Using descriptive names, for example, `totalIncome` rather than `x2`, is almost always a good idea.

[7] R also permits a right-pointing arrow for assignment, as in `2 + 3 -> x`, but its use is uncommon.
[8] *Nonstandard names* may also be used in a variety of contexts, including assignments, by enclosing the names in back-ticks, or in single or double quotes (e.g., `"given name" <- "John"`). Nonstandard names can lead to unanticipated problems, however, and in almost all circumstances are best avoided.

R commands using variables simply substitute the values of the variables for their names:

```
x/2  # equivalent to c(1, 2, 3, 4)/2
  [1] 0.5 1.0 1.5 2.0
(y <- sqrt(x))
  [1] 1.0000 1.4142 1.7321 2.0000
```

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 to make our R code more compact.

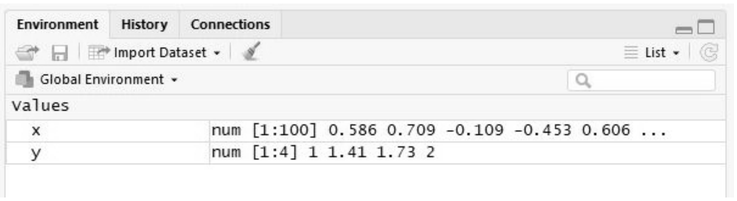
Variables in R are dynamically defined, meaning that we need not tell the interpreter in advance how many values `x` will hold or whether it contains integers, real numbers, character strings, or something else. Moreover, if we wish, we may freely *overwrite* (i.e., redefine) an existing variable, here, `x`:

```
(x <- rnorm(100))  # 100 standard-normal random numbers
    [1]  0.58552882  0.70946602 -0.10930331 -0.45349717  0.60588746
    [6] -1.81795597  0.63009855 -0.27618411 -0.28415974 -0.91932200
   [11] -0.11624781  1.81731204  0.37062786  0.52021646 -0.75053199
   . . .
   [91] -0.96390148 -0.85508251  1.88694694 -0.39181937 -0.98063295
   [96]  0.68733210 -0.50504352  2.15771982 -0.59979756 -0.69454669
```

The `rnorm()` function generates standard-normal random numbers,[9] in this case, 100 of them. Two additional arguments of `rnorm()`, `mean` and `sd`, which are 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 (for details, see `help("rnorm")`). 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; thus, the first value in the second line of output is the sixth element of the vector `x`.

Figure 1.3 shows the contents of the RStudio *Environment* tab after `x` has been overwritten. The *Environment* tab displays a brief summary of objects defined in the *global environment*, currently the numeric vector `x` with 100 values, the first five of which are shown, and the numeric vector `y`. The global environment, also called the user *workspace*, is the region of your computer's memory that holds objects created at the R command prompt. Overwriting `x` with a new value did not change the value of `y`.

[9] Because the values are sampled randomly, when you enter this command you'll get a different result from ours. Random-number generation in R, including how to make the results of random simulations reproducible, is discussed in Section 10.7.



```
Values
  x          num [1:100] 0.586 0.709 -0.109 -0.453 0.606 ...
  y          num [1:4] 1 1.41 1.73 2
```

Figure 1.3 The *Environment* tab shows the name, and an abbreviated version of the value, for all objects that you define in the global environment.

The `summary()` function is an example of a *generic function*: How it behaves depends on its argument. Applied to the numeric vector `x` of 100 numbers sampled from the standard-normal distribution, we get

```
summary(x)

    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  -2.380  -0.590   0.484   0.245   0.900   2.477
```

In this case, `summary(x)` prints the minimum and maximum values of its argument, along with the mean, median, and first and third quartiles (but, curiously, not the standard deviation). Applied to another kind of object, to a data set or a regression model, for example, `summary()` produces different information.[10]

1.2.5 Nonnumeric Vectors

Vectors may also contain nonnumeric values. For example, the command

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

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

defines a *character vector* whose elements are *character strings*. Many R functions work with character data. For example, we may call `paste()` to turn the vector `words` into a single character string:

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

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

The very useful `paste()` function pastes character strings together; the `collapse` argument, as its name implies, collapses the character vector into a single string, separating the elements by the character or characters between the quotation marks, in this case one blank space.[11]

[10] The `args()` and `help()` functions may not be very helpful with generic functions. See Section 1.7 for an explanation of how generic functions in R work.

[11] The `paste()` function is discussed along with other functions for manipulating character data in Section 2.6.

A *logical vector* consists of elements that are either `TRUE` or `FALSE`:

```
(logical.values <- 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,[12] `T` and `F` are not, an omission that we regard as a design flaw in the language.[13] For example, you can perniciously assign `T <- FALSE` and `F <- TRUE` (Socrates was executed for less!). For this reason, we suggest that you avoid the symbols `T` and `F` in your R code and that you also avoid using `T` and `F` as variable names.

There are R functions and operators for working with logical vectors. For example, the `!` ("not") operator negates a logical vector:

```
!logical.values

 [1] FALSE FALSE  TRUE FALSE
```

If we use logical values in arithmetic, R treats `FALSE` as if it were zero and `TRUE` as if it were 1:

```
sum(logical.values)

 [1] 3

sum(!logical.values)

 [1] 1
```

If we create a vector mixing character strings, logical values, and numbers, we produce 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 1:

```
c(10, FALSE, -6.5, TRUE)

 [1] 10.0  0.0 -6.5  1.0
```

These examples illustrate *coercion*: In the first case, we say that the logical and numeric values are *coerced* to character values; in the second case, the logical values are coerced to numbers. In general, coercion in R takes place naturally, and is designed to lose as little information as possible.

[12] For the full set of reserved symbols in R, see `help("Reserved")`.

[13] It would, however, be difficult to make `T` and `F` reserved symbols now, because doing so would break some existing R code.

1.2.6 Indexing Vectors

If we wish to access, perhaps 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] 1.8173
words[2]           # second element
 [1] "be"
logical.values[3] # third element
 [1] FALSE
```

We may also specify a vector of indices:

```
x[6:15]        # elements 6 through 15
  [1] -1.81796  0.63010 -0.27618 -0.28416 -0.91932 -0.11625
  [7]  1.81731  0.37063  0.52022 -0.75053
x[c(1, 3, 5)] # 1st, 3rd, 5th elements
 [1]  0.58553 -0.10930  0.60589
```

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

```
x[-(11:100)] # omit elements 11 through 100
  [1]  0.58553  0.70947 -0.10930 -0.45350  0.60589 -1.81796
  [7]  0.63010 -0.27618 -0.28416 -0.91932
```

The parentheses around `11:100` serve to avoid generating numbers from −11 to 100, which would result in an error. (Try it!) In this case, `x[-(11:100)]` is just a convoluted way of obtaining `x[1:10]`, but negative indexing can be very useful in statistical data analysis, for example, to delete outliers from a computation.

Indexing a vector by a logical vector of the same length selects the elements with `TRUE` indices; for example,

```
v <- 1:4
v[c(TRUE, FALSE, FALSE, TRUE)]
 [1] 1 4
```

Logical values frequently arise through the use of *relational operators*, all of which are *vectorized*, which means that they apply on an elementwise basis to vectors:

`==`	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. Using = where == is intended is a common mistake and can result either in a syntax error or, worse, in an inadvertent assignment, so be careful!

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

`&`	and (vectorized)
`&&`	and (for single left and right operands)
`|`	or (vectorized)
`||`	or (for single left and right operands)
`!`	not (vectorized)

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

The *unvectorized* versions of the *and* (`&&`) and *or* (`||`) operators, included in the table, are primarily useful for writing R programs (see Chapter 10) and are not appropriate for indexing vectors.

An extended example illustrates the use of the comparison and logical operators in indexing:

```
(z <- x[1:10]) # first 10 elements of x
  [1]  0.58553  0.70947 -0.10930 -0.45350  0.60589 -1.81796
  [7]  0.63010 -0.27618 -0.28416 -0.91932
```

```
z < -0.5 # is each element less than -0.5?
   [1] FALSE FALSE FALSE FALSE FALSE  TRUE FALSE FALSE FALSE  TRUE
z > 0.5  # is each element greater than 0.5
   [1]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE FALSE FALSE FALSE
z < -0.5 | z > 0.5  #  < and > are of higher precedence than |
   [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
abs(z) > 0.5  # absolute value, equivalent to last expression
   [1]  TRUE  TRUE FALSE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
z[abs(z) > 0.5] # values of z for which |z| > 0.5
  [1]  0.58553  0.70947  0.60589 -1.81796  0.63010 -0.91932
z[!(abs(z) > 0.5)] # values z for which |z| <= 0.5
  [1] -0.10930 -0.45350 -0.27618 -0.28416
```

The `abs()` function returns the absolute value of its argument. The last of these commands uses the `!` operator to negate the logical values produced by `abs(z) > 0.5` and thus selects the numbers for which the condition is `FALSE`.

A couple of pointers about using the logical and relational 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 the spaces are not *required* around operators, they usually help to clarify R commands.

- Logical operators have lower precedence than relational operators, and so `z < -0.5 | z > 0.5` is equivalent to `(z < -0.5) | (z > 0.5)`. When in doubt, parenthesize!

1.2.7 User-Defined Functions

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

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

Recall that, as shown in the RStudio *Environment* tab, `x` is a vector of 100 standard-normal random numbers, and so this result is the mean of those 100 values.

Were there no `mean()` function, we could nevertheless have calculated the mean straightforwardly using the built-in functions `sum()` and `length()`:

```
sum(x)/length(x)
  [1] 0.2452
```

where the `length()` function returns the number of elements in its argument. To do this repeatedly every time we need to compute the mean of a numeric vector would be inconvenient, and so in the absence of the standard R `mean()` function, we could define our own mean function:[14]

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

- We define a function using the function `function()`.[15] The arguments to `function()`, here just `x`, are the *formal arguments* (also called *dummy arguments*) of the function to be defined, `myMean()`. An *actual argument* will appear in place of the formal argument when the function `myMean()` is called in an R command.
- The remainder of the function definition is an R expression, enclosed in curly braces, specifying the *body* of the function. Typically, the function body is a *compound expression* consisting of several R commands, each on a separate line (or, less commonly, separated by semicolons). The *value returned* by the function is then the value of the last command in the function body.[16] In our simple `myMean()` function, however, the function body consists of a single command.[17]
- The rules for naming functions are 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 the additional argument `na.rm`, which tells the function what to do if some of the elements of `x` are missing.[18]

  If we had chosen to name our function `mean()`, then our `mean()` function, which resides in the global environment, would *shadow* or *mask* the standard `mean()` function (see Section 2.3), and calling `mean()` at the R command prompt would invoke our `mean()` function rather than the standard one. We could in this circumstance restore use of the standard `mean()` function simply by deleting our function from the global environment, via the command `remove("mean")`. You cannot, however, delete a standard R function or a function in a package you have loaded.

[14] When an R command like the definition of the `myMean()` function extends across more than one line, as it is entered into the *Console*, the > prompt changes to +, indicating the continuation of the command. As explained in the Preface, we don't shown the command and continuation prompts in R input displayed in the text. The R interpreter recognizes that a command is to be continued in this manner if it's syntactically incomplete, for example, if there's an opening brace, `{`, unmatched by a closing brace, `}`.

[15] 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.

[16] It is also possible to terminate function execution and explicitly return a value using `return()`. See Chapter 10 for more information about writing R functions.

[17] When the function body comprises a single R command, it's not necessary to enclose the body in braces; thus, we *could* have defined the function more compactly as `myMean <- function(x) sum(x)/length(x)`.

[18] Section 2.3.2 explains how missing data are represented and handled in R.

- In naming functions, we prefer using *camel case*, as in `myMean()`, to separate words in a function name, rather than separating words by periods: for example, `my.mean()`. Periods in function names play a special role in object-oriented programming in R (see Section 1.7), and using periods in the names of ordinary functions consequently invites confusion.[19] Some R users prefer to use *snake case*, which employs underscores to separate words in function and variable names: for example, `my_mean()`.
- Functions, such as `myMean()`, that you create in the global environment are listed in the *Functions* section of the RStudio *Environment* tab.
- We will introduce additional information about writing R functions as required and take up the topic more systematically in Chapter 10.

User-defined functions in R are employed in exactly the same way as the standard R functions. Indeed, most of the standard functions in R are themselves written in the R language.[20] Proceeding with the `myMean()` function,

```
myMean(x)
  [1] 0.2452
y # from sqrt(c(1, 2, 3, 4))
  [1] 1.0000 1.4142 1.7321 2.0000
myMean(y)
  [1] 1.5366
myMean(1:100)
  [1] 50.5
myMean(sqrt(1:100))
  [1] 6.7146
```

As these examples illustrate, there is no necessary correspondence between the name of the formal argument `x` of the function `myMean()` and an actual argument to the function. Function arguments are evaluated by the interpreter, and it is the *value* of the actual argument that is passed to the function, not its name. In the last example, 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 any variables that are defined within a function, are *local* to the function and exist only while the function executes. These *local variables* are distinct from *global variables* of the same names residing in the global environment, which, as we have seen, are listed in the RStudio *Environment* tab.[21]

[19] Nevertheless, largely for historical reasons, many R functions have periods in their names, including standard functions such as `install.packages()`.
[20] 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.
[21] In more advanced use of RStudio, you can pause a function while it is executing to examine variables in its local environment; see Section 10.8.

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 formal argument `x`, but this argument is local to the function `myMean()` and thus did not alter the contents of the global variable `x`, as you can confirm by examining the *Environment* tab.

Our function `myMean()` is used in the same way as standard R functions, and we can consequently use it in defining other functions. For example, the R function `sd()` can compute the standard deviation of a numeric vector. Here's a simple substitute, `mySD()`, which uses `myMean()`:

```
mySD <- function(x){
    sqrt(sum((x - myMean(x))^2)/(length(x) - 1))
}
mySD(1:100)

 [1] 29.011

sd(1:100) # check

 [1] 29.011
```

1.3 Fixing Errors and Getting Help

1.3.1 When Things Go Wrong

Errors can result from bugs in computer software, but much more commonly, they are the fault of the user. No one is perfect, and it is impossible to use a computer without making mistakes. Part of the craft of computing is *debugging*, that is, finding and fixing errors.

- 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 will do what you intend, try it out. You can often debug a command by trying it 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 book, from R help pages, or from textbooks or journal articles can help convince you that a program is working properly.[22]
- 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

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

example, when the interpreter tells you that an object is not found, suspect a typing error that inadvertently produced the name of a nonexistent object, or that you forgot to load a package or to define a variable or a function used in the offending command.

- The source of an error may be subtle, particularly because an R command can generate a sequence (or *stack*) 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'll use the `mySD()` function for computing the standard deviation, defined in Section 1.2.7; to remind ourselves of the definition of this function, and of `myMean()`, which `mySD()` calls, we can print the functions by typing their names (without the parentheses that would signal a function *call*), as for any R objects:[23]

  ```
  mySD

   function(x){
       sqrt(sum((x - myMean(x))^2)/(length(x) - 1))
   }

  myMean

   function(x){
       sum(x)/length(x)
   }
   <bytecode: 0x000000001cbbdb28>
  ```

  We deliberately produce an error by foolishly calling `mySD()` with a nonnumeric argument:

```
letters

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

mySD(letters)

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

  The built-in variable `letters` contains the lowercase letters, and of course, calculating the standard deviation of character data makes no sense. Although the source of the problem is obvious, the error occurs in the `sum()` function, not directly in `mySD()`.[24] The `traceback()` function, called after the offending command, shows the sequence of function calls culminating in the error:

[23] The "`bytecode`" message below the listing of `myMean()` indicates that R has translated the function into a form that executes more quickly.
[24] Were it programmed more carefully, `mySD()` would perform sanity checks on its argument and report a more informative error message when, as here, the argument is nonsensical.

```
traceback()

 2: myMean(x) at #2
 1: mySD(letters)
```

Here, "`at #2`" refers to the second line of the definition of `mySD()`, where `myMean()` is called.

- Not all mistakes 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 investigate suspicious results. It's also generally a bad idea to ignore *warnings*, even though, unlike errors, they don't prevent the completion of a computation.

- If you need to interrupt the execution of a command, you may do so by pressing the Esc (escape) key or by using the mouse to press the *Stop* button at the upper right of the RStudio *Console* pane.[25]

1.3.2 Getting Help and Information

What should you do if the information provided by a call to the `help()` function 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 many thousands of functions in the more than 12,000 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:

- The `apropos()` command searches for currently accessible objects whose names contain a particular character string.[26] For example,

```
apropos("log")

 . . .
 [11] "is.logical"           "log"
 [13] "log10"                "log1p"
 [15] "log2"                 "logb"
 [17] "Logic"                "logical"
 . . .
```

If we're looking for a function to compute logs, this command turns up some relevant results (e.g., `log`, `log10`) and many irrelevant ones.

[25] The *Stop* button only appears *during* a computation.

[26] The `apropos()` function can also search for character patterns called *regular expressions* (which are discussed in Section 2.6).

- Casting a broader net, the `help.search()` command examines the titles and certain other fields in the help pages of all R packages installed in your library, showing the results in the RStudio *Help* tab (which also has its own search box, at the top right of the tab). For example, try the command `help.search("loglinear")` to find functions related to loglinear models (discussed in Section 6.6). 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 `help("RSiteSearch")` for details.

- The CRAN *task views* are documents that describe resources in R for applications in specific areas, such as Bayesian statistics, econometrics, psychometrics, social statistics, and spatial statistics. There are (at the time of writing) more than 30 task views, available via the command `carWeb("taskviews")` (using the `carWeb()` function from the **car** package), directly by pointing your web browser at `https://cran.r-project.org/web/views/`, or in the home screen of the RStudio *Help* tab.

- The command `help(package="`*`package-name`*`")`, for example, `help(package="car")`, displays information about an installed package in the RStudio *Help* tab, including a hyperlinked 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.

- R and RStudio have a number of resources and references available, both locally and on the internet. An index to the key resources can be obtained on the RStudio *Help* tab by clicking on the *Home* icon in the toolbar at the top of the tab.

- Help on R is available on the internet from many other sources. A Google search for `R residualPlot`, for example, will lead to a page at `https://www.rdocumentation.org` for the `residualPlot()` function in the **car** package. The web page `https://www.r-project.org/`

help.html has several suggestions for getting help with R, including information on the R email lists and the StackOverflow@StackOverflow website discussed in the Preface to this book. Also see http://search.r-project.org/.

1.4 Organizing Your Work in R and RStudio and Making It Reproducible

If you organize your data analysis workflow carefully, you'll be able to understand what you did even after some time has elapsed, to reproduce and extend your work with a minimum of effort, and to communicate your research clearly and precisely. RStudio has powerful tools to help you organize your work in R and to make it reproducible. Working in RStudio is surprisingly simple, and it is our goal in this section to help you get started. As you become familiar with RStudio, you may wish to consult the RStudio documentation, conveniently accessible from the *Help* tab, as described at the end of the preceding section, to learn to do more.

We showed in Section 1.2.1 how to interact with the R interpreter by typing commands directly in the *Console*, but that's not generally an effective way to work with R. Although commands typed into the *Console* can be recovered from the *History* tab and by the up- and down-arrow keys at the command prompt, no permanent and complete record of your work is retained. That said, it sometimes makes sense to type *particular* commands directly at the R command prompt—for example, help() and install.packages() commands—but doing so more generally is not an effective way to organize your work.

We'll describe two better strategies here: writing annotated scripts of R commands and writing R Markdown documents.[27]

If you have not already done so, we suggest that you now create an RStudio project named R-Companion, as described in Section 1.1. Doing so will give you access to several files used both in this section and elsewhere in the book.

1.4.1 Using the RStudio Editor With R Script Files

An *R script* is a plain-text document containing a sequence of R commands. Rather than typing commands directly at the R command prompt, you enter commands into the script and execute them from the script, either command by command, several selected commands at once, or the whole script as a unit. Saving the script in a file creates a permanent record of your work that you can use to reproduce, correct, and extend your data analysis.

[27] RStudio is also capable of creating and working with other kinds of documents. For example, we wrote this book in LaTeX using RStudio and the **knitr** package (Xie, 2015, 2018), which support the creation of LaTeX documents that incorporate R code, in the same manner as R Markdown supports the incorporation of R code in a Markdown document.

RStudio makes it easy to create, work with, and manage R scripts. A new script is created by selecting *File* > *New File* > *R Script* from the RStudio menus. The first time that you do this, a new source-editor pane will open on the top left of the RStudio window, and the *Console* will shrink to create space for the new pane. You should now have one tab, called *Untitled1*, in the source pane. When you create a new script in this manner, you should normally save the script in your project directory, using *File* > *Save*, and giving the script a file name ending in the file type .R. RStudio saves R script files as plain-text files.

RStudio recognizes files with extension .R or .r as R scripts and handles these files accordingly, for example by highlighting the syntax of R code (e.g., comments are colored green by default), by automatically indenting continuation lines of multiline R commands as you enter them, and by providing tools for executing R commands.[28] In a subsequent session, you can reopen an existing R script file in several ways: The simplest method is to click on the file's name in the RStudio *Files* tab. Alternatively, you can use the *File* > *Open File* or *File* > *Recent Files* menu. Finally, files that were open in tabs in the RStudio source-editor pane at the end of a session are reopened automatically in a subsequent session.

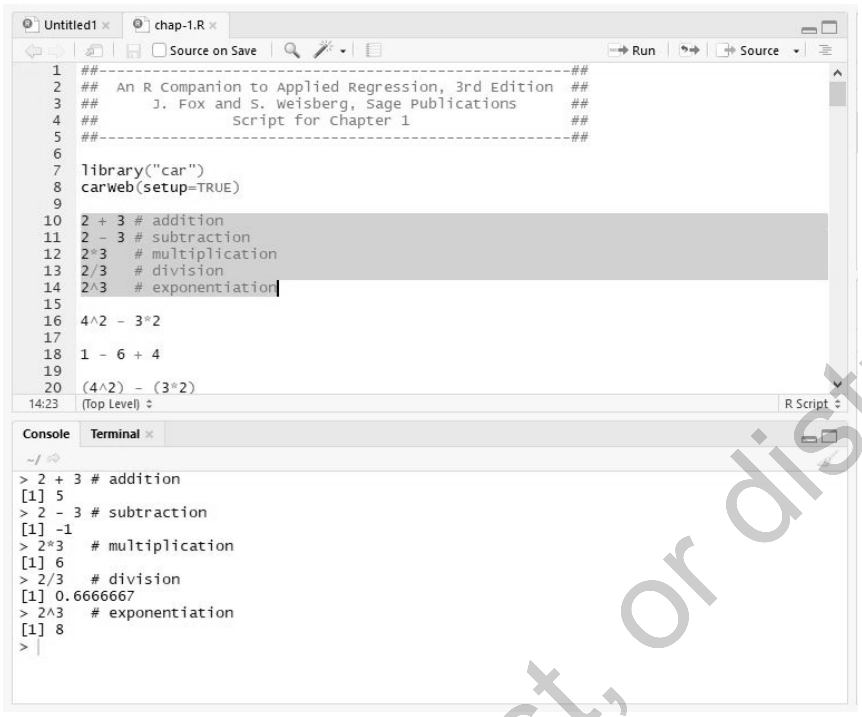
If you liberally sprinkle your R scripts with explanatory comments and notes, you'll be in a better position to understand what you did when you revisit a script at some future date. Scripts provide a simple and generally effective, if crude, means of making your work reproducible.

The file chap-1.R that you downloaded to your project folder in Section 1.1 includes all the R commands used in Chapter 1 of the *R Companion*. Figure 1.4 shows this script open in the RStudio source-editor pane, along with the *Console* pane. As with most editors, you can click the mouse anywhere you like and start typing. You can erase text with the Delete or Backspace key. Common editing gestures are supported, like double-clicking to select a word, triple-clicking to select a whole line, and left-clicking and dragging the mouse across text to select it. In Figure 1.4, we clicked and dragged to select five lines, and then pressed the *Run* button at the top right of the source pane. The selected commands were sent to the R interpreter, as if they had been typed directly at the R command prompt; both the input commands and associated printed output appear in the *Console*, as shown in Figure 1.4.

Some controls in the RStudio editor depend on the type of file that you are editing. After a while, you will likely find these context-dependent changes to be intuitive. For .R files, important editing operations are located in the RStudio *File* and *Edit* menus, and in the toolbar at the top of the file tab, as shown in Figure 1.4.

- To save a file, use the *Save* or *Save as* items in the *File* menu, click on the disk image in the main toolbar at the top of the source-editor pane, or click on the disk image in the toolbar at the top of the document tab.
- To find a text string, or to find and replace text, either click on the spyglass in the toolbar at the top of the document tab, or select *Edit* > *Find* from the RStudio menus. Finding and replacing text works as it does in most editors.

[28] You can customize the RStudio editor in the *Appearance* tab on *Tools* > *Global Profile*



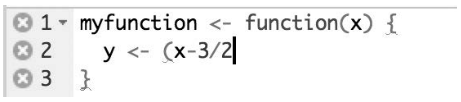
```
##----------------------------------------------------##
##   An R Companion to Applied Regression, 3rd Edition  ##
##       J. Fox and S. Weisberg, Sage Publications      ##
##                 Script for Chapter 1                 ##
##----------------------------------------------------##

library("car")
carWeb(setup=TRUE)

2 + 3 # addition
2 - 3 # subtraction
2*3   # multiplication
2/3   # division
2^3   # exponentiation

4^2 - 3*2

1 - 6 + 4

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

```
> 2 + 3 # addition
[1] 5
> 2 - 3 # subtraction
[1] -1
> 2*3   # multiplication
[1] 6
> 2/3   # division
[1] 0.6666667
> 2^3   # exponentiation
[1] 8
>
```

Figure 1.4 The left-hand panes of the RStudio window. The script files chap-1.R and Untitled1 are in the source-editor pane (at the top). We highlighted several commands in chap-1.R by left-clicking and dragging the mouse over them and caused these commands to be executed by pressing the *Run* button in the toolbar at the top right of the editor tab for the file. The commands and resulting output appear in the R *Console* pane (at the bottom).

- To run selected code, click on the *Run* button at the right of the toolbar above the document tab. To run a single line of code, put the cursor in that line and press *Run*. You can also choose *Code > Run Selected Line(s)* from the RStudio menus.
- If *Source on Save* at the top left of the document tab is checked, then whenever you save the script file, all of its contents are sent to the R interpreter (i.e., the file is "sourced"). Similarly, the *Source* button at the top right of the document tab sends all of the commands in the script to the interpreter but without saving the file. The *Source* button is actually a menu; click on its inverted triangle to see the various options.

All of these actions have keyboard equivalents that you can discover from the menu items or (in most cases) by hovering your mouse over the toolbar icons. Two key-combinations that we use frequently are Ctrl-F (on Windows) or command-



```
myfunction <- function(x) {
  y <- (x-3/2
}
```

Figure 1.5 Snippet from the RStudio source error showing the definition of a function that contains unmatched parentheses. The error is flagged by the x-marks to the left of the lines in the function definition.

F (on macOS), which opens the find/replace tool, and Ctrl-Enter or command-return, which runs the currently selected command or commands. You can open a window with all of RStudio's many keyboard equivalents via the key-combination Alt-Shift-K (or option-shift-K on a Mac).

The RStudio editor has sophisticated tools for writing R code, including tools that are very helpful for finding and correcting errors, such as bugs in user-defined functions.[29] For example, the snippet from the source editor shown in Figure 1.5 contains code defining a function that will not run correctly, as indicated by the (red) x-marks to the left of the numbers of the offending lines. Hovering the mouse over one of the x-marks reveals the problem, an unmatched opening parenthesis. Of course, RStudio can't tell *where* you want the missing closing parenthesis to go. When you insert the matching parenthesis, the error marks go away.

The editor and, for that matter, the *Console* automatically insert and check for matching parentheses, square brackets, curly braces, and quotation marks. The automatic insertion of delimiters can be unnerving at first, but most users eventually find this behavior helpful.[30]

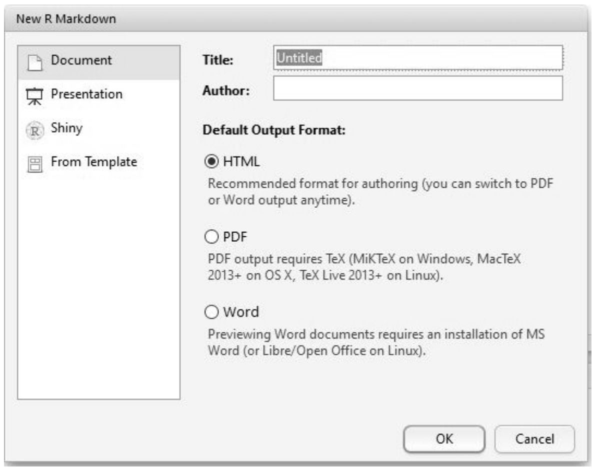
1.4.2 Writing R Markdown Documents

R Markdown documents take script files to a new level by allowing you to mix R commands with explanatory text. Think of an R Markdown document as an R script on steroids. Your R Markdown source document is compiled into an output report evaluating the R commands in the source document to produce easily reproducible results in an aesthetically pleasing form. We believe that for most readers of this book, R Markdown is the most effective means of using R for statistical data analysis.

Markdown, on which R Markdown is based, is a simple, punningly named, *text-markup* language, with simple conventions for adding the main features of a typeset document, such as a title, author, date, sections, bulleted and numbered lists, choices of fonts, and so on. R Markdown enhances Markdown by allowing you to incorporate *code chunks* of R commands into the document. You can press the *Knit* button in the toolbar at the top left of the tab containing an R Markdown document tab to compile the document into a typeset report. When an R

[29] RStudio also incorporates tools for debugging R programs; see Section 10.8.

[30] If automatic insertion of delimiters annoys you, you can turn it off in the *General* tab of the *Tools > Global Options* dialog. Indeed, many features of the RStudio editor can be customized in the *General*, *Code*, and *Appearance* tabs of this dialog, so if you encounter an editor behavior that you dislike, you can probably change it.


New R Markdown

Title: Untitled

Author:

Default Output Format:

● HTML
Recommended format for authoring (you can switch to PDF or Word output anytime).

○ PDF
PDF output requires TeX (MiKTeX on Windows, MacTeX 2013+ on OS X, TeX Live 2013+ on Linux).

○ Word
Previewing Word documents requires an installation of MS Word (or Libre/Open Office on Linux).

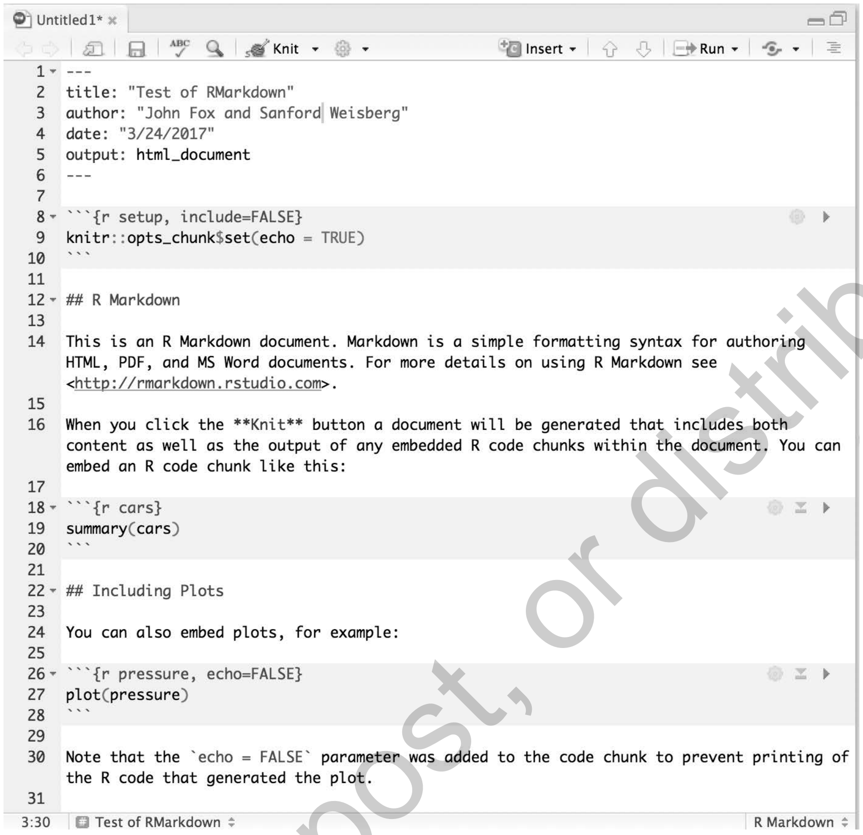
Figure 1.6 The *New R Markdown* dialog box, produced by selecting *File* > *New File* > *R Markdown* from the RStudio menus.

Markdown document is compiled, the R commands in the document are run in an independent R session, and the commands and associated output appear in the report that is produced—just as the commands and output would have appeared in the *Console* or the *Plots* tab if you had typed the R commands in the document into the *Console*. As we will explain, it is also possible to display R text output and graphical output associated with a code chunk *directly* in the R Markdown source document.

R Markdown is sufficiently simple that most users will be able to produce attractively typeset reports using only the instructions we supply here. If you want to learn more about R Markdown, you can select *Help* > *Markdown Quick Reference* from the RStudio menu, which will open the *Markdown Quick Reference* in the RStudio *Help* tab. This document summarizes many of the formatting commands available in Markdown and has a link to more detailed information on the RStudio website. The R Markdown "cheatsheet," from *Help* > *Cheatsheets*, provides similar information in a compact form that is suitable for printing. Finally, for a book-length treatment of R Markdown and other kinds of dynamic R documents, consult Xie (2015).[31]

You can create a new R Markdown document by selecting *File* > *New File* > *R Markdown*... from the RStudio menus, producing the dialog box shown in Figure 1.6. Fill in the *Title* field in the dialog with `Test of RMarkdown` and the *Author* field with your name. Leave the other selections at their defaults, creating

[31] Xie (2015) is written by the author of the **knitr** package and a coauthor of the **rmarkdown** package; these packages are the mechanism that RStudio uses to compile R Markdown documents into reports.



~~~
---
title: "Test of RMarkdown"
author: "John Fox and Sanford Weisberg"
date: "3/24/2017"
output: html_document
---

```{r setup, include=FALSE}
knitr::opts_chunk$set(echo = TRUE)
~~~

R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring
HTML, PDF, and MS Word documents. For more details on using R Markdown see
<http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both
content as well as the output of any embedded R code chunks within the document. You can
embed an R code chunk like this:

```{r cars}
summary(cars)
```

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```

Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of
the R code that generated the plot.
```

Figure 1.7 The skeleton R Markdown file produced by the menu selection *File > New File > R Markdown. . . .*

a document that will produce an HTML (web-page) report.[32]

A skeleton R Markdown source document opens in a tab in the RStudio source-editor pane, as shown in Figure 1.7. We recommend that you immediately rename and save this document as a file of type .Rmd, say RMarkdownTest.Rmd. The source document consists of three components: a *header*, comprising the first six lines; ordinary free-form discursive text, intermixed with simple Markdown formatting instructions; and *code chunks* of commands to be executed by the R interpreter when the source document is compiled into a report.

The document header is filled in automatically and includes title, author, date, and output format fields. You can edit the header as you can edit any other part of the R Markdown document. Consult the R Markdown documentation for more information on the document header.

[32] If you want to produce PDF output, you must first install LaTeX on your computer; instructions for downloading and installing LaTeX are given in the Preface. Once LaTeX is installed, RStudio will find and use it. Another option is to produce a Word document. We discourage you from doing so, however, because you may be tempted to edit the Word document directly, rather than modifying the R Markdown document that produced it, thus losing the ability to reproduce your work.
```

The text component of the document can include simple Markdown formatting instructions. For example, any line that begins with a single # is a major section heading, ## is a secondary section heading, and so on. The *Markdown Quick Reference* lists formatting markup for italics, bold face, bulleted and numbered lists, and so on. Text surrounded by back-ticks, for example, `` `echo = FALSE` ``, is set in a typewriter font, suitable for representing computer code or possibly variable names.

R code chunks appear in gray boxes in an R Markdown document and are demarcated by the line ```` ```{r} ```` at the top of the chunk and the line ```` ``` ```` (three back-ticks) at the bottom. The initial line may also include *chunk options*. You can name a chunk—for example, `setup` in the first chunk in the sample document and `cars` in the second chunk—or supply other options, such as `echo=FALSE`, which suppresses printing the commands in a block in the compiled report, showing only the output from the commands, or `include=FALSE`, which suppresses printing both the commands and associated output. If multiple chunk options are supplied, they must be separated by commas. For more information about R Markdown chunk options, see the webpage at `https://yihui.name/knitr/options/`, or click the gear icon at the top right of a code chunk and follow the link to *Chunk options*.

You can insert a new code chunk in your R Markdown document manually by typing the initial line ```` ```{r} ```` and terminal line ```` ``` ```` or by positioning the cursor at the start of a line and pressing the *Insert* button near the top right of the document tab and selecting *R* from the resulting drop-down list. Click on the gear icon at the top right of a gray code chunk to set some chunk options. The other two icons at the top right of the code chunk allow you to run either all the code chunks *before* the current chunk (the down-pointing triangle) or to run the *current* code chunk (the right-pointing triangle). Hover the mouse cursor over each icon for a "tool tip" indicating the function of the icon.

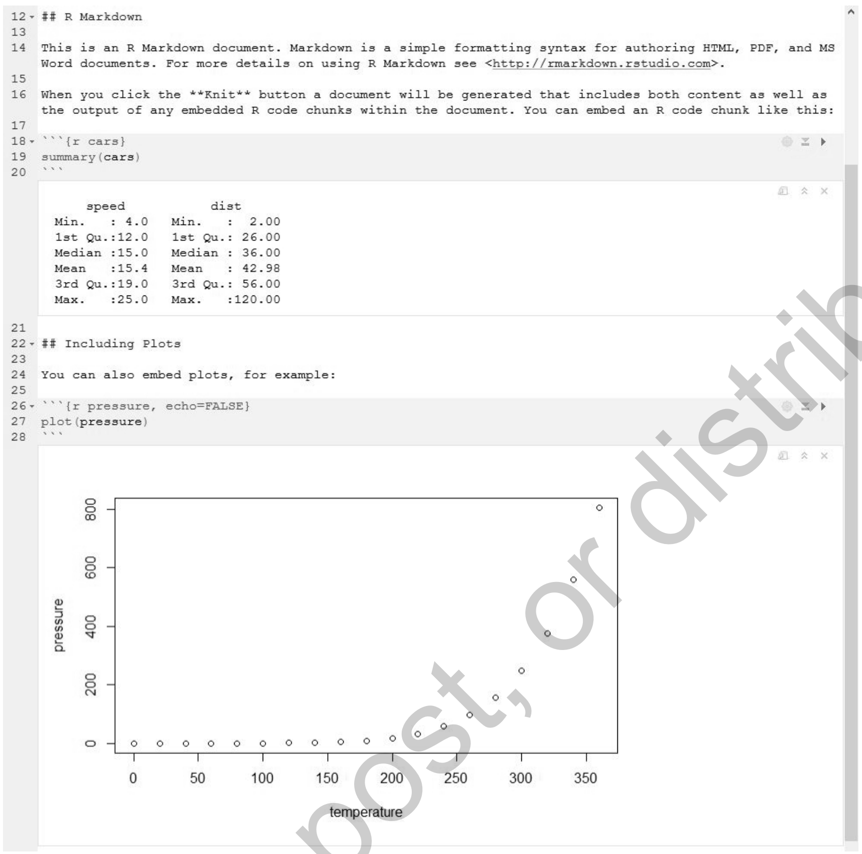
The body of each code chunk consists of commands to be executed by the R interpreter, along with comments beginning with #, which the interpreter will ignore. The various chunk options tell R Markdown how to treat the R input and output in the HTML document that it produces. Almost any R commands may be included in a code chunk.[33]

In addition to code chunks of R commands, you can include *in-line* executable R code, between `` `r `` and another back-tick, `` ` ``. For example, a sentence in your R Markdown document that reads

```
The number of combinations of six items taken two at
a time is `r choose(6, 2)`.
```

would be typeset by evaluating the R command `choose(6, 2)` and replacing the command with its value, `15`—the number of ways of choosing two objects from among six objects, computed by the `choose()` function:

[33] It is best, however, to avoid commands that require direct user intervention; examples of such commands include `identify()`, for interactive point identification in a plot, and `file.choose()` for interactive file selection.



~~~
## R Markdown

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS
Word documents. For more details on using R Markdown see <http://rmarkdown.rstudio.com>.

When you click the **Knit** button a document will be generated that includes both content as well as
the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```{r cars}
summary(cars)
~~~

     speed   dist
 Min.   : 4.0   Min.   : 2.00
 1st Qu.:12.0   1st Qu.: 26.00
 Median :15.0   Median : 36.00
 Mean   :15.4   Mean   : 42.98
 3rd Qu.:19.0   3rd Qu.: 56.00
 Max.   :25.0   Max.   :120.00

Including Plots

You can also embed plots, for example:

```{r pressure, echo=FALSE}
plot(pressure)
```
```

**Figure 1.8** Part of the illustrative R Markdown document (displayed in its entirety in Figure 1.7) as it appears in RStudio. The in-document output was produced by pressing the arrow at the top right of each code chunk to run the code in the chunk.

The number of combinations of six items taken two at a time is 15.

Moreover, if your document assigns values to the variables `n <- 6` and `k <- 2` in a previous code chunk, you could replace `choose(6, 2)` with `chose(n, k)`, and R would substitute the values of these variables correctly into the command.

Turning RMarkdownTest.Rmd into a typeset report can be accomplished effortlessly by pressing the *Knit* button at the top left of the source document tab.[34]

[34] This procedure is essentially effortless only if there are no errors in an R Markdown document. Errors in code chunks typically produce R error messages, and fixing these errors is no different from debugging a script (see Section 1.3.1). You should, however, debug the code chunks interactively *before* compiling the document, as we will explain presently. Markdown errors, in contrast, may cause the document to fail to compile or may result in undesired formatting in the typeset report. Particularly if the document fails to compile, you should examine the output in the *R Markdown* tab, which appears in the lower-right pane in RStudio alongside the *Console* and may contain informative error messages.
```

The resulting report, RMarkdownTest.html, is displayed in RStudio's built-in web-page viewer and is automatically saved in your project directory. R code in the report appears in gray boxes, and associated text output appears in white boxes immediately below the command or commands that produced it. The R > command prompt is suppressed, and each line of output is preceded by two R comment characters, ##. The rationale for formatting commands and output in this manner is that commands can then be copied and pasted directly into R scripts or at the R command prompt without having to remove extraneous >s, while output preceded by ## will be treated as R comments. If you prefer to include the command prompts and suppress the hash-marks in the compiled document, you can alter the first code chunk in the document to read

~~~
```{r setup, include=FALSE, eval=FALSE}
knitr::opts_chunk$set(echo=TRUE, prompt=TRUE, comment="")
```
~~~

When you develop an R script, you normally execute commands as you write them, correcting and elaborating the commands in a natural manner. R Markdown source documents support a similar style of work: After you type a new command into a code chunk, you can execute it by pressing the *Run* button at the top of the document tab and choosing *Run Selected Line(s)* from the drop-down list, pressing the key-combination Control-Enter (as you would for a script), or selecting *Code* > *Run Selected Line(s)* from the RStudio menus. The command is executed in the R *Console*, and any output from the command—both text and graphics—appears as well in the source-editor tab for the R Markdown document. This in-document output is independent of the web-page report that's produced when you compile the whole document. You can, consequently, maximize the editor pane, covering the *Console*, and still see the output. We recommend this approach to developing R Markdown documents interactively.

You can also execute an entire code chunk by selecting *Run Current Chunk* from the *Run* button list or via *Code* > *Run Current Chunk*. Alternatively, and most conveniently, each code chunk has small icons at the upper right of the chunk; the right-pointing arrow runs the code in the chunk, displaying output immediately below. See Figure 1.8 for an example.

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

In this section, we use standard R functions along with functions in the **car** package in a typical linear regression problem. An R Markdown document that repeats all the R commands, but with minimal explanatory text, is in the file Duncan.Rmd that you downloaded to your R-Companion project in Section 1.1.[35] Following the instructions in Section 1.4.2, you can typeset this document by clicking on

[35] If you choose not to create an R-Companion project, you can still obtain this file by the command `carWeb(file="Duncan.Rmd")`, which downloads the file to your RStudio working directory.

Duncan.Rmd in the RStudio *Files* tab to open the document in the source editor and then clicking the *Knit* button in the toolbar at the top of the document tab.

The data for this example are in the **carData** package, which is automatically loaded when you load the **car** package, as we've done earlier in this chapter. Working with your own data almost always requires the preliminary work of importing the data from an external source, such as an Excel spreadsheet or a plain-text data file. After the data are imported, you usually must perform various data management tasks to prepare the data for analysis. Importing and managing data in R are the subject of the next chapter.

The Duncan data set that we use in this section was originally analyzed by the sociologist Otis Dudley Duncan (1961). The first 10 lines of the `Duncan` *data frame* are printed by the `head()` command:

```
head(Duncan, n=10)

            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
reporter      wc     67        87       52
engineer    prof     72        86       88
```

```
dim(Duncan)

[1] 45  4
```

A data frame is the standard form of a rectangular data set in R, a two-dimensional array of data with columns corresponding to variables and rows corresponding to cases or observations (see Section 2.3). The first line of the output shows the names for the variables, `type`, `income`, `education`, and `prestige`. Each subsequent line contains data for one case. The cases are occupations, and the first entry in each line is the name of an occupation, generally called a *row name*. There is no variable in the data frame corresponding to the row names, but you can access the row names by the command `row.names(Duncan)` or `rownames(Duncan)`. The `dim()` ("`dim`ensions") command reveals that the `Duncan` data frame has 45 rows (cases) and four columns (variables). Because this is a small data set, you could print it in full in the *Console* simply by entering its name, `Duncan`, at the R command prompt. You can also open the data frame in a spreadsheet-like tab in the RStudio source-editor pane by entering the command `View(Duncan)`.[36]

The definitions of the variables in Duncan's data set are given by `help("Duncan")` and are as follows:

[36] The `View()` function shouldn't be used in an R Markdown document, however, because it creates output outside the *Console* and *Plots* tab. Although the RStudio data set viewer *looks* like a spreadsheet, you can't edit a data frame in the viewer.

- `type`: Type of occupation, `prof` (professional and managerial), `wc` (white-collar), or `bc` (blue-collar)
- `income`: Percentage of occupational incumbents in the 1950 U.S. Census who earned $3,500 or more per year (about $36,000 in 2017 U.S. dollars)
- `education`: Percentage of occupational incumbents in 1950 who were high school graduates (which, were we cynical, we would say is roughly equivalent to a PhD in 2017)
- `prestige`: Percentage of respondents in a social survey who rated the occupation as "good" or better in prestige

The variable `type` has character strings for values rather than numbers, and is called a *factor* with the three *levels* or categories `"prof"`, `"wc"`, and `"bc"`.[37] The other variables are numeric. Duncan used a linear least-squares regression of `prestige` on `income` and `education` to predict the prestige of occupations for which the income and educational scores were known from the U.S. Census but for which there were no direct prestige ratings. He did not use occupational `type` in his analysis.

This is a small data frame in an era of "big data," but, for several reasons, we think that it makes a nice preliminary example:

- Duncan's use of multiple regression to analyze the data was unusual at the time in sociology, and thus his analysis is of historical interest.
- Duncan's methodology—using a regression for a subset of occupations to impute prestige scores for all occupations—is still used to create occupational socioeconomic status scales and consequently is not *just* of historical interest.
- The story of Duncan's regression analysis is in part a cautionary tale, reminding us to check that our statistical models adequately summarize the data at hand.

The generic `summary()` function has a *method* that is appropriate for data frames. As described in Section 1.7, generic functions adapt their behavior to their arguments. Thus, a function such as `summary()` may be used appropriately with many different kinds of objects. This ability to reuse the same generic function for many similar purposes is one of the strengths of R. When applied to the `Duncan` data frame, `summary()` produces the following output:

```
summary(Duncan)

   type        income        education        prestige
 bc  :21   Min.   : 7.0   Min.   :  7.0   Min.   : 3.0
 prof:18   1st Qu.:21.0   1st Qu.: 26.0   1st Qu.:16.0
 wc  : 6   Median :42.0   Median : 45.0   Median :41.0
           Mean   :41.9   Mean   : 52.6   Mean   :47.7
           3rd Qu.:64.0   3rd Qu.: 84.0   3rd Qu.:81.0
           Max.   :81.0   Max.   :100.0   Max.   :97.0
```

[37] For efficiency of storage, values of a factor are actually coded numerically as integers, and the corresponding level names are recorded as an *attribute* of the factor. See Chapter 2 and Section 4.7 for more on working with factors.

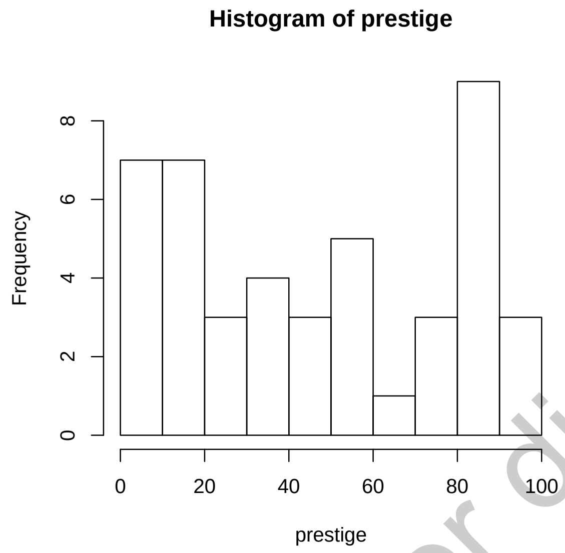


Figure 1.9 Histogram for `prestige` in Duncan's data frame.

The function counts the number of cases in each level of the factor `type` and reports the minimum, maximum, median, mean, and the first and third quartiles for each numeric variable, `income`, `education`, and `prestige`.

1.5.1 Examining the Data

A sensible place to start any data analysis, including a regression analysis, is to visualize the data using a variety of graphical displays. Figure 1.9, for example, shows a histogram for the response variable `prestige`, produced by a call to the `hist()` function:

```
with(Duncan, hist(prestige))
```

The `with()` function makes the `prestige` variable in the `Duncan` data frame available to `hist()`.[38] The `hist()` function doesn't return a visible value in the R console but rather is used for the *side effect* of drawing a graph, in this case a histogram.[39] Entering the `hist()` command from an R script displays the histogram in the RStudio *Plots* tab; entering the command in an R Markdown document

[38] The general format of a call to `with()` is `with(`*`data-frame, R-command`*`)`, where *`R-command`* could be a *compound expression* enclosed in braces, `{ }`, and comprising several commands, each command on its own line or separated from other commands by semicolons. We discuss managing and using data in data frames in Chapter 2.

[39] Like all R functions, `hist()` *does* return a result; in this case, however, the result is *invisible* and is a list containing the information necessary to draw the histogram. To render the result visible, put parentheses around the command: `(with(Duncan, hist(prestige)))`. Lists are discussed in Section 2.4.

displays the graph directly in the document and in the HTML report compiled from the R Markdown document.

The distribution of `prestige` appears to be bimodal, with cases stacking up near the boundaries, as many occupations are either low prestige, near the lower boundary, or high prestige, near the upper boundary, with relatively fewer occupations in the middle bins of the histogram. Because `prestige` is a percentage, this behavior is not altogether unexpected. Variables such as this often need to be transformed, perhaps with a logit (log-odds) or similar transformation, as discussed in Section 3.4. Transforming `prestige` turns out to be unnecessary in this problem.

Before fitting a regression model to the data, we should also examine the distributions of the predictors `education` and `income`, along with the relationship between `prestige` and each predictor, and the relationship between the two predictors.[40] The `scatterplotMatrix()` function in the **car** package associated with this book allows us to conveniently examine these distributions and relationships.[41]

```
scatterplotMatrix( ~ prestige + education + income,
    id=list(n=3), data=Duncan)
```

The `scatterplotMatrix()` function uses a *one-sided formula* to specify the variables to appear in the graph, where we read the formula `~ prestige + education + income` as "plot `prestige` and `education` and `income`." The `data` argument tells `scatterplotMatrix()` where to find the variables.[42] The argument `id=list(n=3)` tells `scatterplotMatrix()` to identify the three most unusual points in each panel.[43] We added this argument after examining a preliminary plot of the data. Using a script or typing the `scatterplotMatrix()` command directly into the *Console* causes the graph to be shown in the RStudio *Plots* tab. You can view a larger version of the graph in its own window by pressing the *Zoom* button at the top of the *Plots* tab. As explained in Section 1.4.2, if you're working in an R Markdown document, you can display the graph directly in the source document.

The scatterplot matrix for `prestige`, `education`, and `income` appears in Figure 1.10. The variable names in the diagonal panels label the axes. The scatterplot in the upper-right-hand corner, for example, has `income` on the horizontal axis and `prestige` on the vertical axis. By default, *nonparametric density estimates*, using an adaptive-kernel estimator, appear in the diagonal panels, with a *rug-plot* ("one-dimensional scatterplot") at the bottom of each panel, showing the location of the data values for the corresponding variable.[44] There are several lines on each scatterplot:

[40] We will ignore the additional predictor `type`, which, as we mentioned, didn't figure in Duncan's analysis of the data.
[41] Because we previously loaded the **car** package for access to the `Duncan` data frame, we do not need to do so again.
[42] We'll encounter formulas again when we specify a regression model for Duncan's data (in Section 1.5.2), and the topic will be developed in detail in Chapter 4 on linear models, particularly in Section 4.9.1.
[43] Point identification in the **car** package is explained in Section 3.5.
[44] Other choices are available for the diagonal panels, including histograms. We discuss `scatterplotMatrix()` and other graphics functions in the **car** package for exploring data in Section 3.3.

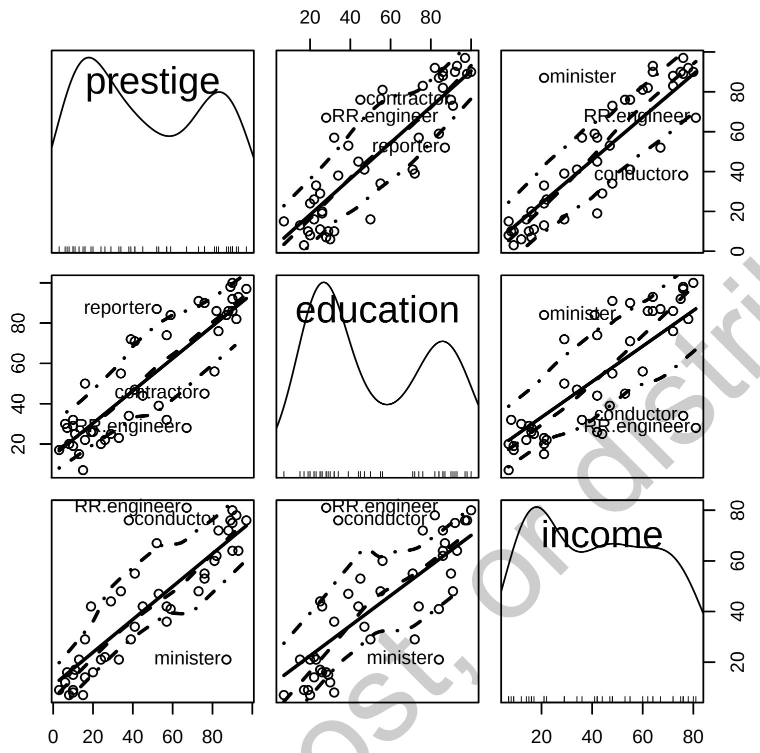


Figure 1.10 Scatterplot matrix for `prestige`, `education`, and `income` in Duncan's data, identifying the three most unusual points in each panel. Nonparametric density estimates for the variables appear in the diagonal panels, with a rug-plot (one-dimensional scatterplot) at the bottom of each diagonal panel.

- The solid line shows the marginal linear least-squares fit for the regression of the vertical-axis variable (*y*) on the horizontal-axis variable (*x*), ignoring the other variables.
- The central broken line is a nonparametric regression smooth, which traces how the average value of *y* changes as *x* changes without making strong assumptions about the form of the relationship between the two variables.[45]
- The outer broken lines represent smooths of the conditional variation of the *y* values given *x* in each panel, like running quartiles.

Like `prestige`, `education` appears to have a bimodal distribution. The distribution of `income`, in contrast, is perhaps best characterized as irregular. The pairwise relationships among the variables seem reasonably linear, which means

[45] Scatterplot smoothers in the **car** package are discussed in Section 3.2.1.

that as we move from left to right across the plot, the average *y* values of the points more or less trace out a straight line. The scatter around the regression lines appears to have reasonably constant vertical variability and to be approximately symmetric.

In addition, two or three cases stand out from the others. In the scatterplot of `income` versus `education`, ministers are unusual in combining relatively low `income` with a relatively high level of `education`, and railroad conductors and engineers are unusual in combining relatively high levels of `income` with relatively low `education`. Because `education` and `income` are predictors in Duncan's regression, these three occupations will have relatively high *leverage* on the regression coefficients. None of these cases, however, are outliers in the *univariate* distributions of the three variables.

1.5.2 Regression Analysis

Duncan was interested in how `prestige` is related to `income` and `education` in combination. We have thus far addressed the univariate distributions of the three variables and the pairwise or marginal relationships among them. Our plots don't directly address the *joint* dependence of `prestige` on `education` and `income`. Graphs for this purpose will be discussed in Section 1.5.3.

Following Duncan, we next fit a linear least-squares regression to the data to model the joint dependence of `prestige` on the two predictors, under the assumption that the relationship of `prestige` to `education` and `income` is additive and linear:

```
(duncan.model <- lm(prestige ~ education + income, data=Duncan))

Call:
lm(formula = prestige ~ education + income, data = Duncan)

Coefficients:
(Intercept)    education       income
     -6.065        0.546        0.599
```

Like the `scatterplotMatrix()` function, the `lm()` (linear model) function uses a *formula* to specify the variables in the regression model, and the `data` argument to tell the function where to find these variables. The formula argument to `lm()`, however, has two sides, with the response variable `prestige` on the left of the tilde (`~`). The right-hand side of the model formula specifies the predictor variables in the regression, `education` and `income`. We read the formula as "`prestige` is regressed on `education` and `income`."[46]

The `lm()` function returns a *linear-model object*, which we assign to `duncan.model`. The name of this object is arbitrary—any valid R name would do. Enclosing the command in parentheses causes the assigned object to be printed, in this instance displaying a brief report of the results of the regression. The `summary()` function produces a more complete report:

[46] You'll find much more information about linear-model formulas in R in Chapter 4, particularly Section 4.9.1.

```
summary(duncan.model)

 Call:
 lm(formula = prestige ~ education + income, data = Duncan)

 Residuals:
    Min     1Q Median     3Q    Max
 -29.54  -6.42   0.65   6.61  34.64

 Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept)  -6.0647     4.2719   -1.42     0.16
 education     0.5458     0.0983    5.56  1.7e-06 ***
 income        0.5987     0.1197    5.00  1.1e-05 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 13.4 on 42 degrees of freedom
 Multiple R-squared:  0.828,       Adjusted R-squared:  0.82
 F-statistic:  101 on 2 and 42 DF,  p-value: <2e-16
```

Both `education` and `income` have large regression coefficients in the `Estimate` column of the coefficient table, with small two-sided *p*-values in the column labeled `Pr(>|t|)`. For example, holding `education` constant, a 1% increase in higher `income` earners is associated on average with an increase of about 0.6% in high `prestige` ratings.

R writes very small and very large numbers in scientific notation. For example, `1.1e-05` is to be read as 1.1×10^{-5}, or 0.000011, and `2e-16` $= 2 \times 10^{-16}$, which is effectively zero.

If you find the "statistical-significance" asterisks that R prints to the right of the *p*-values annoying, as we do, you can suppress them, as we will in the remainder of the *R Companion*, by entering the command:[47]

```
options(show.signif.stars=FALSE)
```

Linear models are described in much more detail in Chapter 4.

1.5.3 Regression Diagnostics

To assume that Duncan's regression in the previous section adequately summarizes the data does not make it so. It is therefore wise after fitting a regression model to check the fit using a variety of graphs and numeric procedures. The standard R distribution includes some facilities for *regression diagnostics*, and the **car** package substantially augments these capabilities.

The `"lm"` object `duncan.model` contains information about the fitted regression, and so we can employ the object in further computations beyond producing a

[47] It's convenient to put this command in your .Rprofile file so that it's executed at the start of each R session, as we explained in the Preface (page xxii). More generally, the `options()` function can be used to set a variety of global options in R; see `?options` for details.

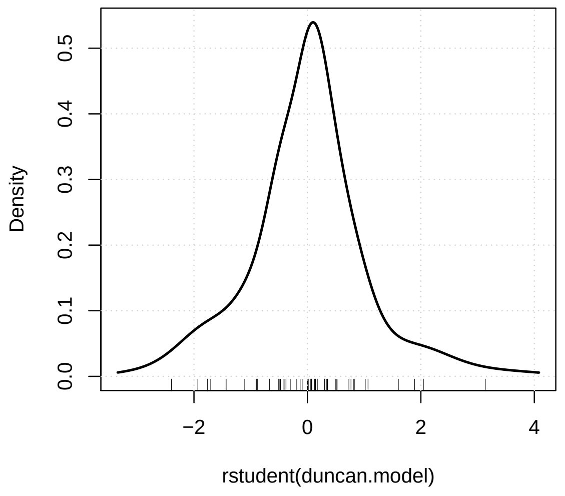


Figure 1.11 Nonparametric density estimate for the distribution of the Studentized residuals from the regression of `prestige` on `education` and `income`.

printed summary of the model. More generally, the ability to manipulate statistical models as objects is a strength of R. The `rstudent()` function, for example, uses some of the information in `duncan.model` to calculate *Studentized residuals* for the model. A nonparametric density estimate of the distribution of Studentized residuals, produced by the `densityPlot()` function in the **car** package and shown in Figure 1.11, is unremarkable:

```
densityPlot(rstudent(duncan.model))
```

Observe the sequence of operations here: `rstudent()` takes the linear-model object `duncan.model`, previously computed by `lm()`, as an argument, and returns the Studentized residuals as a result, then passes the residuals as an argument to `densityPlot()`, which draws the graph. This style of command, where the result of one function becomes an argument to another function, is common in R.

If the errors in the regression are normally distributed with zero means and constant variance, then the Studentized residuals are each t-distributed with $n - k - 2$ degrees of freedom, where k is the number of coefficients in the model, excluding the regression constant, and n is the number of cases. The generic `qqPlot()` function from the **car** package, which makes *quantile-comparison plots*, has a method for linear models:

```
qqPlot(duncan.model, id=list(n=3))

  minister   reporter contractor
         6          9         17
```

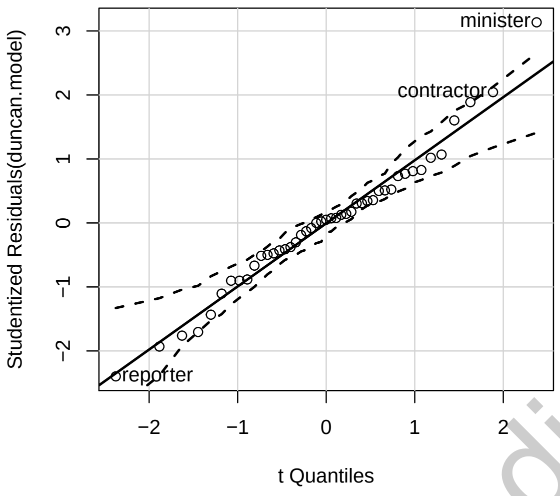


Figure 1.12 Quantile-comparison plot for the Studentized residuals from the regression of `prestige` on `education` and `income`. The broken lines show a bootstrapped pointwise 95% confidence envelope for the points.

The resulting quantile-comparison plot is shown in Figure 1.12. The `qqPlot()` function extracts the Studentized residuals and plots them against the quantiles of the appropriate *t*-distribution. If the Studentized residuals are *t*-distributed, then the plotted points should lie close to a straight line. The solid comparison line on the plot is drawn by default by robust regression. The argument `id=list(n=3)` identifies the three most extreme Studentized residuals, and `qqPlot()` returns the names and row numbers of these cases.

In this case, the residuals pull away slightly from the comparison line at both ends, suggesting that the residual distribution is a bit heavy-tailed. This effect is more pronounced at the upper end of the distribution, indicating a slight positive skew.

By default, `qqPlot()` also produces a bootstrapped pointwise 95% confidence envelope for the Studentized residuals that takes account of the correlations among them (but, because the envelope is computed pointwise, does not adjust for simultaneous inference). The residuals stay nearly within the boundaries of the envelope at both ends of the distribution, with the exception of the occupation `minister`.[48] A test based on the largest (absolute) Studentized residual, using the `outlierTest()` function in the **car** package, however, suggests that the residual for ministers is not terribly unusual, with a Bonferroni-corrected *p*-value of 0.14:

[48] The bootstrap procedure used by `qqPlot()` generates random samples, and so the plot that you see when you duplicate this command will not be identical to the graph shown in the text.

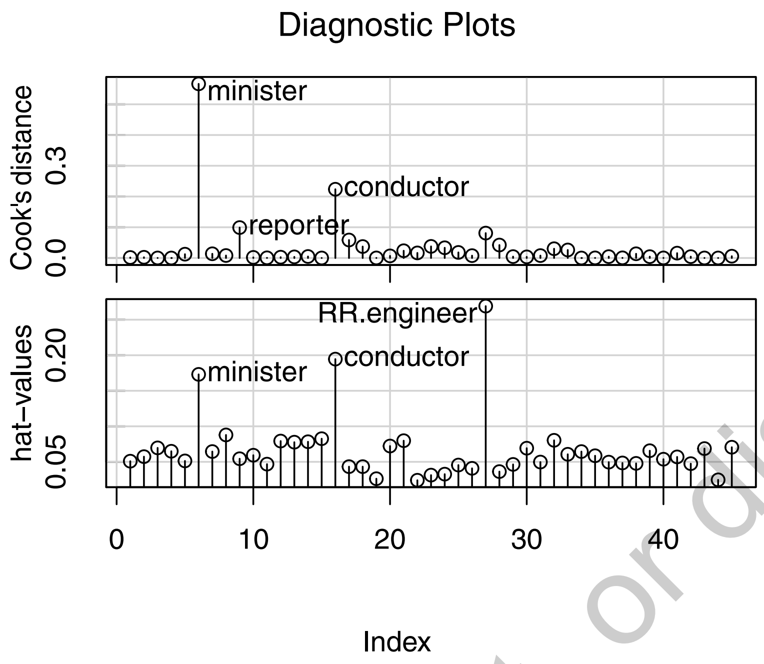


Figure 1.13 Index plots of Cook's distances and hat-values, from the regression of `prestige` on `income` and `education`.

```
outlierTest(duncan.model)

 No Studentized residuals with Bonferonni p < 0.05
 Largest |rstudent|:
          rstudent unadjusted p-value Bonferonni p
 minister   3.1345          0.0031772      0.14297
```

We proceed to check for high-leverage and influential cases by using the `influenceIndexPlot()` function from the **car** package to plot *hat-values* (Section 8.3.2) and *Cook's distances* (Section 8.3.3) against the case indices:

```
influenceIndexPlot(duncan.model, vars=c("Cook", "hat"),
    id=list(n=3))
```

The two index plots are shown in Figure 1.13. We ask to identify the three biggest values in each plot.

Because the cases in a regression can be *jointly* as well as individually influential, we also examine *added-variable plots* for the predictors, using the `avPlots()` function in the **car** package (Section 8.2.3):

```
avPlots(duncan.model,
    id=list(cex=0.75, n=3, method="mahal"))
```

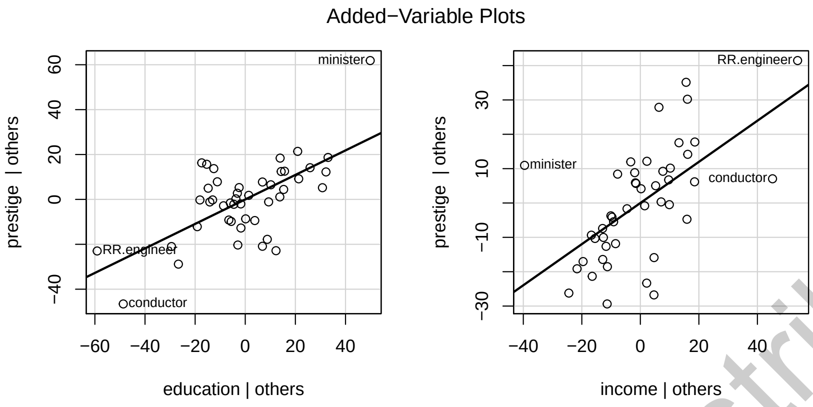


Figure 1.14 Added-variable plots for `education` and `income` in Duncan's occupational-prestige regression.

The `id` argument, which has several components here, customizes identification of points in the graph:[49] `cex=0.75` (where `cex` is a standard R argument for "character expansion") makes the labels smaller, so that they fit better into the plots; `n=3` identifies the three most unusual points in each plot; and `method="mahal"` indicates that unusualness is quantified by Mahalanobis distance from the center of the point-cloud.[50]

Each added-variable plot displays the *conditional*, rather than the marginal, relationship between the response and one of the predictors. Points at the extreme left or right of the plot correspond to cases that have high leverage on the corresponding coefficient and consequently are potentially influential. Figure 1.14 confirms and strengthens our previous observations: We should be concerned about the occupations `minister` and `conductor`, which work jointly to increase the `education` coefficient and decrease the `income` coefficient. Occupation `RR.engineer` has relatively high leverage on these coefficients but is more in line with the rest of the data.

We next use the `crPlots()` function, also in the **car** package, to generate *component-plus-residual plots* for `education` and `income` (as discussed in Section 8.4.2):

```
crPlots(duncan.model)
```

The component-plus-residual plots appear in Figure 1.15. Each plot includes a least-squares line, representing the regression plane viewed edge-on in the direction of the corresponding predictor, and a *loess* nonparametric-regression smooth.[51] The purpose of these plots is to detect nonlinearity, evidence of which is slight here.

[49] See Section 3.5 for a general discussion of point identification in **car**-package plotting functions.
[50] Mahalanobis distances from the center of the data take account of the standard deviations of the variables and the correlation between them.
[51] See Section 3.2 for an explanation of scatterplot smoothing in the **car** package.

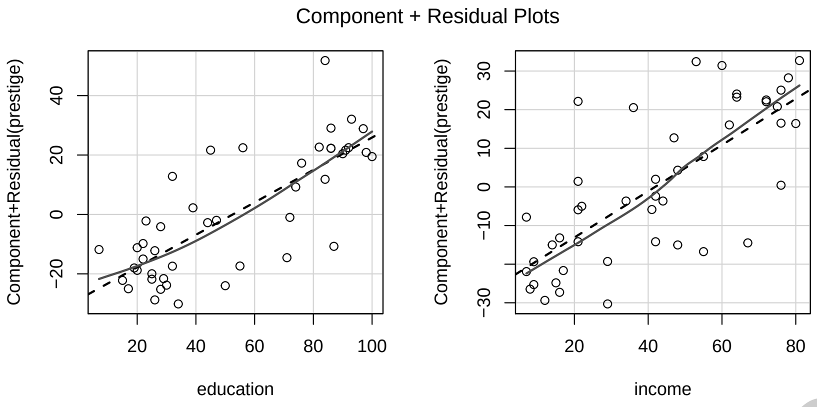


Figure 1.15 Component-plus-residual plots for `education` and `income` in Duncan's occupational-prestige regression. The solid line in each panel shows a loess nonparametric-regression smooth; the broken line in each panel is the least-squares line.

Using the `ncvTest()` function in the **car** package (Section 8.5.1), we compute score tests for nonconstant variance, checking for an association of residual variability with the fitted values and with *any* linear combination of the predictors:

```
ncvTest(duncan.model)

 Non-constant Variance Score Test
 Variance formula: ~ fitted.values
 Chisquare = 0.3811, Df = 1, p = 0.537

ncvTest(duncan.model, var.formula= ~ income + education)

 Non-constant Variance Score Test
 Variance formula: ~ income + education
 Chisquare = 0.6976, Df = 2, p = 0.706
```

Both tests yield large *p*-values, indicating that the assumption of constant variance is tenable.

Finally, on the basis of the influential-data diagnostics, we try removing the cases `minister` and `conductor` from the regression:

```
whichNames(c("minister", "conductor"), Duncan)

  minister conductor
         6        16

duncan.model.2 <- update(duncan.model, subset=-c(6, 16))
summary(duncan.model.2)

 Call:
 lm(formula = prestige ~ education + income, data = Duncan,
     subset = -c(6, 16))
```

```
Residuals:
   Min     1Q Median     3Q    Max
-28.61  -5.90   1.94   5.62  21.55

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  -6.4090     3.6526   -1.75   0.0870 .
education     0.3322     0.0987    3.36   0.0017 **
income        0.8674     0.1220    7.11  1.3e-08 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 11.4 on 40 degrees of freedom
Multiple R-squared:  0.876,       Adjusted R-squared:  0.87
F-statistic:  141 on 2 and 40 DF,  p-value: <2e-16
```

Rather than respecifying the regression model from scratch with `lm()`, we refit it using the `update()` function, removing the two potentially problematic cases via the `subset` argument to `update()`. We use the `whichNames()` function from the **car** package to remind us of the indices of the two cases to be removed, `minister` (Case 6) and `conductor` (Case 16).

The `compareCoefs()` function, also from the **car** package, is convenient for comparing the estimated coefficients and their standard errors across the two regressions fit to the data:

```
compareCoefs(duncan.model, duncan.model.2)

Calls:
1: lm(formula = prestige ~ education + income, data =
  Duncan)
2: lm(formula = prestige ~ education + income, data =
  Duncan, subset = -c(6, 16))

            Model 1 Model 2
(Intercept)   -6.06   -6.41
SE             4.27    3.65

education    0.5458  0.3322
SE           0.0983  0.0987

income        0.599   0.867
SE            0.120   0.122
```

The coefficients of `education` and `income` changed substantially with the deletion of the occupations `minister` and `conductor`. The `education` coefficient is considerably smaller and the `income` coefficient considerably larger than before. Further work (not shown, but which we invite the reader to duplicate) suggests that removing occupations `RR.engineer` (Case 27) and `reporter` (Case 9) does not make much of a difference to the results.

Chapter 8 has much more extensive information on regression diagnostics in R, including the use of various functions in the **car** package.

1.6 R Functions for Basic Statistics

The focus of the *R Companion* is on using R for regression analysis, broadly construed. In the course of developing this subject, we will encounter, and indeed already have encountered, a number of R functions for basic statistical methods (`mean()`, `hist()`, etc.), but the topic is not addressed systematically.

Table 1.1 shows the names of some standard R functions for basic data analysis. The R help system, through `?` or `help()`, provides information on the usage of these functions. Where there is a substantial discussion of a function in a later chapter in the *R Companion*, the location of the discussion is indicated in the column of the table marked *Reference*. The table is not meant to be complete.

1.7 Generic Functions and Their Methods*

Many of the most commonly used functions in R, such as `summary()`, `print()`, and `plot()`, produce different results depending on the arguments passed to the function.[52] For example, the `summary()` function applied to different columns of the `Duncan` data frame produces different output. The summary for the variable `type` is the count in each level of this factor,[53]

```
summary(Duncan$type)

  bc prof   wc
  21   18    6
```

while for a numeric variable, such as `prestige`, the summary includes the mean, minimum, maximum, and several quantiles:

```
summary(Duncan$prestige)

    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
     3.0    16.0    41.0    47.7    81.0    97.0
```

Similarly, the commands

```
summary(Duncan)

    type        income       education        prestige
  bc  :21   Min.   : 7.0   Min.   :  7.0   Min.   : 3.0
  prof:18   1st Qu.:21.0   1st Qu.: 26.0   1st Qu.:16.0
  wc  : 6   Median :42.0   Median : 45.0   Median :41.0
            Mean   :41.9   Mean   : 52.6   Mean   :47.7
```

[52] The generic `print()` function is invoked implicitly and automatically when an object is printed by typing the name of the object at the R command prompt or in the event that the object returned by a function isn't assigned to a variable. The `print()` function can also be called explicitly, however.

[53] `Duncan$type` selects the variable `type` from the `Duncan` data frame. Indexing data frames and other kinds of objects is discussed in detail in Section 2.4.4.

Table 1.1 Some R functions for basic statistical methods. All functions are in the standard R packages; chapter references are to the *R Companion*.

Method	R *Function(s)*	*Reference*
Basic Graphs		
histogram	`hist()`	Chapter 3
stem-and-leaf display	`stem()`	Chapter 3
boxplot	`boxplot()`	Chapter 3
scatterplot	`plot()`	Chapter 3
time-series plot	`ts.plot()`	
Numerical Summaries		
mean	`mean()`	
median	`median()`	
quantiles	`quantile()`	
extremes	`range()`	
variance	`var()`	
standard deviation	`sd()`	
covariance matrix	`var()`, `cov()`	
correlations	`cor()`	
Probability		
normal density, distribution, quantiles, and random numbers	`dnorm()`, `pnorm()`, `qnorm()`, `rnorm()`	Chapter 3
t density, distribution, quantiles, and random numbers	`dt()`, `pt()`, `qt()`, `rt()`	Chapter 3
chi-square density, distribution, quantiles, and random numbers	`dchisq()`, `pchisq()`, `qchisq()`, `rchisq()`	Chapter 3
F density, distribution, quantiles, and random numbers	`df()`, `pf()`, `qf()`, `rf()`	Chapter 3
binomial probabilities, distribution, quantiles, and random numbers	`dbinom()`, `pbinom()`, `qbinom()`, `rbinom()`	Chapter 3
generating random samples	`sample()`, `rnorm()`, etc.	
Basic Linear Models		
simple regression	`lm()`	Chapter 4
multiple regression	`lm()`	Chapter 4
analysis of variance	`aov()`, `lm()`, `anova()`	Chapter 4
Contingency Tables		
contingency tables	`xtabs()`, `table()`	Chapter 6
printing tables	`ftable()`	Chapter 6
percentage tables	`prop.table()`	Chapter 6
Simple Hypothesis Tests		
t-tests for means	`t.test()`	
tests for proportions	`prop.test()`, `binom.test()`	
chi-square test for independence	`chisq.test()`	Chapter 6
various nonparametric tests	`friedman.test()`, `kruskal.test()`, `wilcox.test()`, etc.	

```
             3rd Qu.:64.0   3rd Qu.: 84.0   3rd Qu.:81.0
             Max.   :81.0   Max.   :100.0   Max.   :97.0
```

and

```
summary(lm(prestige ~ education + income, data=Duncan))

 Call:
 lm(formula = prestige ~ education + income, data = Duncan)

 Residuals:
    Min     1Q Median     3Q    Max
 -29.54  -6.42   0.65   6.61  34.64

 Coefficients:
             Estimate Std. Error t value Pr(>|t|)
 (Intercept)  -6.0647     4.2719   -1.42     0.16
 education     0.5458     0.0983    5.56  1.7e-06 ***
 income        0.5987     0.1197    5.00  1.1e-05 ***
 ---
 Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

 Residual standard error: 13.4 on 42 degrees of freedom
 Multiple R-squared:  0.828,        Adjusted R-squared:  0.82
 F-statistic:  101 on 2 and 42 DF,  p-value: <2e-16
```

produce output appropriate to these objects—in the first case by summarizing each column of the `Duncan` data frame and in the second by returning a standard summary for a linear regression model.

Enabling the same *generic function*, such as `summary()`, to be used for many purposes is accomplished in R through an *object-oriented programming* technique called *object dispatch*. The details of object dispatch are implemented differently in the S3 and S4 object systems, so named because they originated in Versions 3 and 4, respectively, of the original S language on which R is based. There is yet another implementation of object dispatch in R for the more recently introduced system of *reference classes* (sometimes colloquially termed "R5").

Almost everything created in R is an *object*, such as a numeric vector, a matrix, a data frame, a linear regression model, and so on.[54] In the S3 object system, described in this section and used for most R object-oriented programs, each object is assigned a *class*, and it is the class of the object that determines how generic functions process the object. We won't take up the S4 and reference-class object systems in this book, but they too are class based and implement (albeit more complex) versions of object dispatch.

[54] Indeed, everything in R that is *returned* by a function is an object, but some functions have *side effects* that create nonobjects, such as files and graphs.

The `class()` function returns the class of an object:

```
class(Duncan$type)

 [1] "factor"

class(Duncan$prestige)

 [1] "integer"

class(Duncan)

 [1] "data.frame"
```

The `lm()` function, to take another example, creates an object of class `"lm"`:

```
duncan.model <- lm(prestige ~ education + income, data=Duncan)
class(duncan.model)

 [1] "lm"
```

Generic functions operate on their arguments indirectly by calling specialized functions, referred to as *method functions* or, more compactly, as *methods*. Which method is invoked typically depends on the class of the first argument to the generic function.[55]

For example, the generic `summary()` function has the following definition:

```
summary

 function (object, ...)
 UseMethod("summary")
 <bytecode: 0x000000001d03ba78>
 <environment: namespace:base>
```

As for any object, we print the definition of the `summary()` function by typing its name (without the parentheses that would *invoke* rather than *print* the function). The generic function `summary()` has one required argument, `object`, and the special argument `...` (the ellipses) for additional arguments that could vary from one `summary()` method to another.[56]

When `UseMethod("summary")` is called by the `summary()` generic, and the first (`object`) argument to `summary()` is of class `"lm"`, for example, R searches for a method function named `summary.lm()`, and, if it is found, executes the command `summary.lm(object, ...)`. It is, incidentally, perfectly possible to call `summary.lm()` directly; thus, the following two commands are equivalent (as the reader can verify):

```
summary(duncan.model)
summary.lm(duncan.model)
```

[55] In contrast, in the S4 object system, method dispatch can depend on the classes of more than one argument to a generic function.

[56] You can disregard the last two lines of the output, which indicate that the function has been compiled into *byte code* to improve its efficiency, something that R does automatically, and that it resides in the namespace of the **base** package, one of the standard R packages that are loaded at the start of each session.

Although the generic summary() function has only one explicit argument, the method function summary.lm() has additional arguments:

```
args("summary.lm")

function (object, correlation = FALSE, symbolic.cor = FALSE,
    ...)
NULL
```

Because the arguments correlation and symbolic.cor have default values (FALSE, in both cases), they need not be specified. Thus, for example, if we enter the command summary(duncan.model, correlation=TRUE), the argument correlation=TRUE is absorbed by ... in the call to the generic summary() function and then passed to the summary.lm() method, causing it to print a correlation matrix for the coefficient estimates.

In this instance, we can call summary.lm() directly, but most method functions are hidden in (not "exported from") the *namespaces* of the packages in which the methods are defined and thus cannot normally be used directly.[57] In any event, it is good R form to use method functions only indirectly through their generics.

Suppose that we invoke the hypothetical generic function fun(), defined as

```
fun <- function(x, ...){
    UseMethod("fun")
}
```

with real argument obj of class "cls": fun(obj). If there is no method function named fun.cls(), then R looks for a method named fun.default(). For example, objects belonging to classes without summary() methods are summarized by summary.default(). If, under these circumstances, there is *no* method named fun.default(), then R reports an error.

We can get a listing of all currently accessible methods for the generic summary() function using the methods() function, with hidden methods flagged by asterisks:[58]

```
methods(summary)

 [1] summary,ANY-method            summary,diagonalMatrix-method
 [3] summary,sparseMatrix-method   summary.Anova.mlm*
 [5] summary.aov                   summary.aovlist*
 [7] summary.aspell*               summary.boot*
. . .
[97] summary.varFunc*              summary.varIdent*
[99] summary.varPower*
see '?methods' for accessing help and source code
```

[57] For example, the summary() method summary.boot(), for summarizing the results of bootstrap resampling (see Section 5.1.3), is hidden in the namespace of the **car** package. To call this function directly to summarize an object of class "boot", we could reference the function with the unintuitive package-qualified name car:::summary.boot(), but calling the unqualified method summary.boot() directly won't work.

[58] The first three method functions shown, with commas in their names, are S4 methods.

These methods may have different arguments beyond the first, and some method functions, for example, `summary.lm()`, have their own help pages: `?summary.lm`.

You can also determine what generic functions have currently available methods for objects of a particular class. For example,

```
methods(class="lm")

  [1] add1                alias               anova
  [4] Anova               avPlot              Boot
  [7] bootCase            boxCox              case.names
 . . .
 [70] summary             variable.names      vcov
 see '?methods' for accessing help and source code
```

Method selection is slightly more complicated for objects whose class is a vector of more than one element. Consider, for example, an object returned by the `glm()` function for fitting generalized linear models (anticipating a logistic-regression example developed in Section 6.3.1):[59]

```
mod.mroz <- glm(lfp ~ ., family=binomial, data=Mroz)
class(mod.mroz)

 [1] "glm" "lm"
```

If we invoke a generic function with `mod.mroz` as its argument, say `fun(mod.mroz)`, then the R interpreter will look first for a method named `fun.glm()`. If a function by this name does not exist, then R will search next for `fun.lm()` and finally for `fun.default()`. We say that the object `mod.mroz` is of *primary class* `"glm"` and *inherits* from class `"lm"`.[60] Inheritance supports economical programming through generalization.[61]

[59] The `.` on the right-hand side of the model formula indicates that the response variable `lfp` is to be regressed on *all* of the other variables in the `Mroz` data set (which is accessible because it resides in the **carData** package).

[60] If the class vector of an object has more than two elements, then the classes are searched sequentially from left to right.

[61] S3 inheritance can also get us into trouble if, for example, there is no function `fun.glm()` but `fun.lm()` exists and is inappropriate for `mod.mroz`. In a case such as this, the programmer of `fun.lm()` should be careful also to create a function `fun.glm()`, which calls the default method or reports an error, as appropriate.